Electronic Version
Stylesheet Version v1.1.1

Description

Concurrent Processing Memory

Cross Reference to Related Applications

[0001] The present application claims the benefit of Provisional Application Number 60/320250 filed June 6, 2003 by Chengpu Wang.

Copyright Statement

Background of Invention

[0003] In the past 40 years, the semiconductor industry has been dictated by Moore's law, which says for every one and half years, the density of semiconductor devices doubles. Moore's law has also applied to CPU speed in a similar

fashion. However, in recent years, the semiconductor industry has slowed down and deviated noticeably from the Moore's law, e.g. the increase of the clock speed of CPU can no longer keep the same pace. Also, the industry faces two major technology challenges for further size reduction: (A) the transition from classical circuit to quantum circuit, and (B) the transition from far-field (wave) manufacture technology to near-field (nano) manufacture technology. At this moment an important question is: Are our computers fast enough?

[0004] The majority of our computers, including PC, Unix, Macintosh, and most embedded computers, are bus-sharing computers, in which there is: (A) a memory unit that stores instructions and data, (B) a processing unit that executes the instructions one after another, to process the data, and (C) a bus unit that connects the two. At MHz or even GHz of clock rate, and even with multiple CPUs within the processing unit, our bus-sharing computers seem quite fast for solving most serial problems which contains sequence of instructions, yet they are ill equipped when dealing with parallel problems such as searching and ordering database, processing image, and modeling involving space, mainly due to the following

reasons:

[0005] (1) The parallel nature of the problem is different from the serial way in which the problem is solved in bus-sharing computers. In a parallel problem, a procedure is applied independently to each item of an array. The collection of such applications can be carried out concurrently. Yet a bus-sharing computer can only carry out them one after another. The drawback is two fold: (A) The amount of data could be huge, e.g., even a common digital camera contains million of pixels. If a same procedure has to be repeated for each array item, it is a very slow solution. For an example, to process a photo taken by a common digital camera, a bus-sharing computer has to repeat a same procedure of a parallel problem at least millions times. (B) Each application of the procedure contains many same operations on the same data, and a bus-sharing computer has to repeat every one of them for each different datum. Thus, it is also a very inefficient solution. For an example, every pair of neighboring data has to be summed multiple times in any neighborhood averaging scheme.

[0006] (2) The large amount of required data transfer for carrying out the serial solution for a parallel problem will boggle down the bus unit in a bus-sharing computer. Actually,

the bus unit is already normally much slower than the processing unit, e.g., in PC, it has been always about 5 times slower for the past ten years. The speed of a bus-sharing computer is usually determined by the speed at which the bus unit can supply instructions and data from the memory unit to the processing unit. This is called a bus bottleneck problem. Trying to cope with this bus bottleneck is already the major task of a modern CPU, e.g. costing about 70% of die area of a Pentium III CPU. Flushing the bus unit of a bus-sharing computer with a lot of repeated instructions and repeated data when solving a parallel problem serially can only make the matter much worse. For an example, the simplest neighborhood averaging of a digital camera photo in a bus-sharing computer requires tens of millions times of pixel data transfer, and all of them are repeated. This adds a lot of stress to the bus unit of the bus-sharing computer.

[0007] The above drawbacks of the bus-sharing computer originate from the separation of (A) the processing and (B) the storing of instructions and data. With the currently achievable semiconductor size and new developments in silicon integration, it is quite desirable to merge the processing and the storing of instructions and data into one

unit. The end of the Moore's law actually provides development possibilities in other dimensions.

[0008] Still, it is not the time to dismiss our bus-sharing computers yet. In addition to their well known advantages of maturity and ubiquity, and amazing abilities for serial problems, bus-sharing computers have one hidden advantage: they fit our Human logic well. Our Human logic is based on induction and deduction, both of which are serial in nature. We only deal with parallel problems as one of the steps of our serial problems. The bus-sharing computers have the architecture that guarantees the serial execution of instructions, and provides bases for proper synchronization between multiple threads of serial executions. Even the reconfigurable systems, such as PLD and FPGA, which are frequently associated with parallel data processing, are mostly configured in programs, which comprise serial descriptive instructions and are processed by bus-sharing computers using serial instructions.

[0009] Another hidden advantage of our bus-sharing computers is that they can have a powerful processing unit that can do almost everything. On the other hand, any solution for parallel problems based on massive parallel processing can not be universal to make economical sense. It is justi-

fied to have one or a few very complicated CPUs for one computer. It is probably not justified to have one very complicated CPU for every datum in a large pool of data.

[0010] So a fast and efficient solution to our parallel problems may call for a device that: (A) integrates seamlessly with a bus-sharing architecture; (B) is controlled by the processing unit of the bus-sharing architecture and is part of the memory unit; (C) is limited to the application of parallel problems only; (D) stores the data for the parallel problem; (E) processes the data locally near each datum; (F) solves the parallel problem using massive parallel algorithm, such as SIMD (Single-Instruction Multiple-Data) in particular; and (G) has minimal impact on the bus unit of the bus-sharing architecture. Or in another word, what we need is a smart memory for each particular kind of parallel problems.

[0011] Information relevant to attempts to build memory with some internal processing power can be found in U.S. Patent No 6,460,127, 6,404,439, 6,711,665, 6,275,920, 4,215,401, 4,739,474, 6,073,185, 5,809,322, 5,717,943, 5,710,932, 5,546,343, 5,421,019, 5,134,711, 5,095,527, 5,038,282, 6,049,859, 6,173,388, 5,752,068, 5,729,758, 5,590,356, 5,555,428, 5,418,915, 5,175,858, 4,992,933,

and 4,775,952. However, each one of these references suffers from one or more of the following disadvantages: (A) Not pin-compatible or function-compatible with a conventional random access memory; (B) Not able to be used in a memory unit of a conventional bus-sharing architecture; (C) Not able to accomplish tasks by itself of required complexity for most common parallel problems such as sorting and sum; (D) requiring a lot reconfiguration effort when switching tasks; and (E) requiring redesigning of existing computer architectures.

[0012] For the foregoing reasons, there is a need to build smart memories that is: (A) pin compatible with a conventional random access memory; (B) function compatible with a conventional random access memory; (C) comprising a SIMD (Single-Instruction Multiple-Data) processing architecture inside; (D) requiring no or little external bus activities to solve the parallel problem for which the smart memory is designed for; (E) switching between different tasks instantly, (F) variable in scope of capability; and (G) is practical to be implemented.

SUMMARY OF INVENTION

[0013] The present invention is directed to an apparatus that satisfies this need for a smart memory. This apparatus is

called concurrent processing memory, or simply CP memory.

[0014] The CP memory is pin compatible with a conventional random access memory. It needs only difference of one extra pin, called a command input pin, from a conventional random access memory. The command input pin can actually be connected as an address pin as if the CP memory is a random access memory of a larger capacity.

[0015] When the command input pin is negatively asserted, the CP memory behaves exactly like a conventional random access memory, containing an array of addressable registers for storing and retrieving data through an external bus comprising address bus, data bus and control bus.

[0016] The CP memory is also a SIMD (Single-Instruction Multiple-Data) machine for solving parallel problems, containing identical memory elements: (A) each of which preferably comprises at least one addressable registers, possibly other registers, and some processing power, and (B) all of which can simultaneously execute a same instruction independently from each other. The concurrent processing power means great reduction of the required instruction cycles for parallel problems. The processing power within the CP memory means reduction, in most

cases great reduction, of the need to use the external bus to transfer data.

[0017] When the command input pin is positively asserted, the CP memory treats the content of the external bus as an instruction. Since the command input pin is connected as a pin for address bus, to a user of the CP memory, sending instruction and getting result is like storing and retrieving data using a special address in a conventional random access memory. In this way, a CP memory can be used anywhere a conventional random access memory can be used, including in any bus-sharing computer.

[0018] A memory element of a CP memory only executes an instruction when it is activated. The CP memory instantly activates all memory elements whose element addresses are: (A) no less than a start address, (B) no more than an end address, and (C) an integer increment of the carry number starting from the start address. In another word, the activated elements form a lattice that is instantly changeable. The lattice structure is analogous with the data array structure which is common to all parallel problems. This guarantees quick task switching, no matter how many memory elements need to be activated or inactivated between tasks.

[0019] The CP memory is actually a family name that comprises CP memories of various scopes, for solving different kinds of parallel problems. Among them, in the order of increased complexity of the memory element, are: (A) content movable memory, (B) content searchable memory, (C) content comparable memory, (D) database memory, (E) 1D math memory, and (F) 2D math memory. The content searchable memory and the content comparable memory are collectively referred as content matchable memory. The 1D math memory and 2D math memory are collectively referred as math memory.

[0020] The CP memory is constructed using standard digital circuitry technology. Still, several device components of the CP memory have been invented also using standard digital circuitry technology, such as carry pattern generator, parallel shifter, all-line decoder, parallel comparator, general decoder, range decoder, multi-channel multiplexer, and multi-channel demultiplexer.

Brief Description of Drawings

[0021] *FIG. 1*: Complex system structure of a complex CP Memory.

[0022] *FIG. 2*: Connecting a CP memory to an external bus.

[0023] *FIG. 3*: Connecting two CP memories together and to an external bus.

[0024] *FIG. 4*: Circuit diagram of a 3-digit 8-input/output parallel left shifter.

[0025] *FIG. 5*: Circuit diagram of a 3-input 8-output all-line decoder.

[0026] *FIG. 6*: Logic for activating general decoder bit outputs.

[0027] *FIG. 7*: Structure diagram of a content movable memory element.

[0028] *FIG. 8a*: Structure diagram of a content searchable memory element.

[0029] *FIG. 8b*: Structure diagram of a content comparable memory element.

[0030] *FIG. 9*: Circuit diagram of a 4-bit parallel comparator.

[0031] *FIG. 10*: Symbols for standard and simplified multiple input AND gate.

[0032] *FIG. 11*: Circuit diagram of a 4-bit parallel adder.

[0033] *FIG. 12*: Structure diagram of a 4-bit parallel counter using adders in binary tree construct.

[0034] *FIG. 13*: Circuit diagram of a 4-bit parallel adder for parallel counter.

[0035] *FIG. 14*: Circuit diagram of a 3-bit parallel counter using

A/D technology.

[0036] *FIG. 15*: Structure diagram of a 6-bit parallel counter scaled up from 3-bit parallel counters.

[0037] *FIG. 16*: Circuit diagram of an 8-input 4-channel multiplexer.

[0038] *FIG. 17*: Circuit diagram of an 8-output 4-channel demultiplexer.

[0039] *FIG. 18*: Structure diagram of a memory element for math memory.

[0040] *FIG. 19*: General cases of disorder for global moving sorting algorithm.

[0041] *FIG. 20*: Algorithm flow diagram for 1-D sum.

[0042] *FIG. 21*: Algorithm flow diagram for 2-D sum.

[0043] *FIG. 22*: Algorithm flow diagram for 1-D template matching.

[0044] *FIG. 23*: Algorithm flow diagram for 2-D template matching.

[0045] *FIG. 24*: (4 * 3) super lattice for detecting line with slope of (3/4).

[0046] *FIG. 25a*: A set of lines whose pixel spans are exactly 7 in walking distance.

[0047] *FIG. 25b*: A set of lines whose pixel spans are about 5 in

real distance.

[0048] *FIG. 26*: Log(N) long range connectivity.

[0049] *FIG. 27a*: 2-D super-lattice connectivity.

[0050] *FIG. 27b*: 3-D super-lattice connectivity.

[0051] *FIG. 28*: Logic diagram of parallel divider.

[0052] *FIG. 29*: Function diagram of a concurrent processing memory, which is the overview of the invention.

DETAILED DESCRIPTION

BACKWARD COMPATIBILITY

[0053] FIG. 1 shows a structure overview of a most complicated CP memory on the system level, which can be turned into other family members in the CP memory family by deleting components form it, as described later in this Description.

[0054] Except a command bit input *101*, a CP memory has the same external bus connection *102* for an external bus as a conventional random access memory. The external bus comprises address bus, data bus, and control bus.

[0055] The address bus is usually wider than a memory's external connection to address bus. For a conventional random access memory, the address bus bits which are not con-

nected with the memory's external bus connection to the address bus are assigned address bits. Each memory has an assigned address which is unique for the memory. When the assigned address bits equals the assigned address, an enable bit input, which is one of the memory's external bus connection to control bus, is positively asserted to activate the memory. For a CP memory, the least significant bit of the assigned address bits is connected to the command input bit of a CP memory, while the rest bits are assigned address bits. Thus, a CP memory requires twice of address space than what it contains in its addressable registers. Other assigned address bit can also be connected to the command input bit of a CP memory, with a larger address space needed.

[0056] The data bus is usually 2^M fold byte wide, in which M is an unsigned integer, while each addressable register inside a memory is often byte wide. If a memory's external connections to data bus are byte wide, the M least significant bits of the address bus select the byte portion of the data bus to be connected to the CP memory's external connection to data bus, using a multiplexer/demultiplexer, in the same manner as a conventional random access memory.

[0057] FIG. 2 shows how a byte-wide CP memory *301* is connected with the address bus and the data bus of an external bus, whose data bus *310* is two-byte wide. The least significant portion *303* of the address bus *302* is connected to the memory's external bus connection to address bus. The next address bus bit *304* is connected with the memory's command input bit. When the rest address bits *305* contain a value that equals the assigned address *308* for the memory, the memory is activated through its enable bit input *307*, which is one of the memory's external bus connections to control bus. The least significant bit *306* of the address bus *302*, which is also connected to the memory's external bus connection to the address bus, selects to connect either the lower portion *311*, or the higher portion *312* of the data bus *310* to the memory's external bus connection to data bus*314* through a multiplexer/demultiplexer *313*.

[0058] The CP memory's external bus connections to the other bits of the control bus are the same as those of a random access memory. The control bus of an external bus provides power and ground, instructs the memory for either a storing or a retrieving operation, and provides synchronization and handshake with other devices which are also

connected to the same external bus.

[0059] If the address space is not a concern, a CP memory may have more than one command bit to connect to the address bits, to increase the bandwidth of transferring instructions. Some bus standards have dedicated control and arbitration bits to control the connected devices. Accordingly, the CP memory may have additional command bits to take advantages of the situation.

Exclusive Access

[0060] In FIG. 1, when the command bit input *101* is negatively asserted, the CP memory behaves exactly like a conventional random access bus. The address bus of the external bus *102* specifies a register address for one of the addressable registers *106* within the CP memory; the register address is sent to the input/output control unit *103*, and then to the register control unit *104*, which exclusively activates the corresponding addressable register at the register address through exclusive connections *107* to each of all the addressable registers. The control bus of the external bus *102* specifies either a storing operation or a retrieving operation to the CP memory. For a storing operation, the data is sent from the data bus of the external bus *102* to the input/output control unit *103*, then to the ex-

clusive bus *105*, and then to the exclusively activated addressable register. For a retrieve operation, the data is sent from the exclusively activated addressable register, to the exclusive bus *105*, then to the input/output control unit *103*, and then to the data bus of the external bus *102*. A CP memory may use the same logic and the same hardware for exclusive access as a random access memory.

Concurrent Instructing

[0061] The CP memory is also a SIMD machine, containing identical memory elements *108*, each of which preferably comprises at least one addressable registers *106*, possibly other registers, an enable bit input*111*, an optional match bit output *112*, and some processing power.

[0062] When the command bit input *101* is positively asserted, the CP memory treats the content of the external bus *102* as an instruction. Since the command input pin *101* is connected as an address bus bit, to a user of a CP memory, sending instruction and getting result is like storing and retrieving data with a conventional random access memory when a particular address bit is positively asserted. Within CP memory, the instruction is then translated by the input/output control unit *103*, and broadcasted to all the memory elements *108* concurrently

through a concurrent bus *109*. In addition to instruction, the concurrent bus *109* may also broadcast data to all the memory elements *108*. The concurrent bus *109* is exclusively written by the input/output control unit *103*, and concurrently read by multiple memory elements *108*.

[0063] Each memory element *108* has a unique element address. The input/output control unit *103* sends a start address, an end address, and a carry number to a general decoder *110*, which, through enable bit inputs *111* exclusively to each of all the memory elements *108*, activates all the memory elements *108* whose element addresses are: (A) no less than the start address, (B) no more than the end address, and (C) an integer increment of the carry number starting from the start address. All the enabled memory elements receive and execute a same instruction with a same data parameter from the concurrent bus *109*. The start address, end address, and carry number are all parameters as part of instructions to the CP memory.

[0064] As described later, the carry number needs not to exceed the square root of the total bit output count of the general decoder. For a content movable memory or a content searchable memory, it is a constant of 1.

[0065] The data for majority parallel problems are in the format

of array. Using the above activation rules, an item may be held by a same number of memory elements which have consecutive element addresses, or a memory element may hold a same number of items. For simplicity of the following discussion, each memory element may hold one item, and the other two cases can be treated similarly.

[0066] It is possible that each of all bit outputs of the general decoder is connected to a dedicated bit storage cell *115*, such as a flip-flop, and the bit storage cell connects to the enable bit input of the corresponding memory element *111*. One use of the bit storage cell *115* is to separate the general decoder from active duty of activating memory elements when the general decoder *110*, parallel counter and priority encoder *113* are configured as a parallel divider, as described later. The other use of the bit storage cell *115* is to put additional constraint on the activation of memory elements, such as acting as a filter for a 2D image pattern which has irregular shape.

[0067] Like a conventional static random access memory, the execution of an instruction by a CP memory may take the same amount of time as storing or retrieving data with an addressable register. Like a conventional dynamic random access memory, the execution of an instruction by a CP

memory may take longer time, or even variable time, and the input/output control unit *103* may use standard asynchronous means for signaling the termination of instruction execution, such as interrupt, wait states, or predefined content change of the external bus *102*, or simply require a predefined wait period before receiving another instruction from the external bus *102*.

[0068] Each register inside a memory element is identified by a register number, so that it can be referred in an instruction to the memory element. The assignment of register number satisfies the following conditions: (1) the set of register numbers is identical for all of the memory elements; (2) the registers which have the same register number are functionally equivalent within their memory elements respectively, and (3) the register number for an addressable register is between zero and the value of one less than the count of the addressable registers within each memory element. Thus, the register address of each addressable register *106* comprises: (1) the element address of the memory element *108* which contains the addressable register *106*; and (2) the register number to identify the addressable register *106* within the memory element. If the register number is used as the lower por-

tion for the register address, all functionally equivalent registers within all memory elements form a continuous register address range, which is convenient for task switching such as using direct memory access.

CONCURRENT MATCHING

[0069] Each activated memory element *108* of a CP memory can have internal states. If the internal state matches a requirement, which may have been sent to the memory elements by the concurrent bus *109*, the memory element positively asserts its match bit output*112* exclusively to a priority encoder *113*, which outputs to the input/output control unit *103* either the highest or the lowest element address of the memory element which is in the required state. The priority of the priority encoder is controlled by the input/output control unit *103*. Alternatively, each match bit output*112* for the memory element may exclusively connect to a parallel counter *113*, which outputs the total count of the memory element which is in the matched state to the input/output control unit *103*. Both priority encoder and parallel counter may also be used.

[0070] Each memory element may have a storage bit to save the binary value of the match bit output, so that it can be used for subsequent state definition, or state definition

which involves neighboring memory elements.

Local Connectivity

[0071] The physically neighboring memory elements have adjacent element addresses. In a one-dimensional CP memory, except the two boundary memory elements, each of which has either lowest or highest element address, each of all the memory elements has two neighboring memory elements whose element address is either immediately lower or immediately higher than the element address of the memory element itself. In a two-dimensional CP memory, each memory element is on the node of a square lattice; the two perpendicular lattice directions are the X and the Y directions; the element address is partitioned into X and Y addresses; and except boundary memory elements; each of all memory elements has a pair of neighboring memory elements along the X direction, and another pair of neighboring memory elements along the Y direction.

[0072] The neighboring memory elements may be connected through neighborhood connection *114* so that each memory element shows a universal content of at least one of its registers, which is called the neighboring register, to all of its neighbors.

[0073] A CP memory may contain additional external connections to the neighboring registers of the boundary memory, so that several CP memories can be connected and used as one large CP memory. FIG. 3 shows how to connect two CP memories together, each of which has been connected to an external bus as described in FIG. 2, using the additional external connections to the neighboring registers of the boundary memory elements *315*

INSTRUCTION KERNEL

[0074] A CP memory is controlled by the external bus, which is connected and controlled by the processing unit of a computer. An instruction kernel may interface between a CP memory and an external bus, to translate instructions for the instruction kernel into instructions for the CP memory, not unlike translating the instructions for a processor into micro-kernel instructions within the processor. The instruction kernel could be: (1) an instruction kernel inside the input/output unit of the CP memory, (2) an embedded microcontroller between the CP memory and the external bus, or (3) a software driver that manages the CP memory.

[0075] The instructions for the instruction kernel are more complex, and probably more capable than the instructions for

the memory elements. For an example, in math memory, the multiplication and division instructions for the instruction kernel may be translated into a series of addition, subtraction, and shifting instructions for the memory elements. The instruction kernel may contain resources such as memory, registers, and/or accumulator to carry out the instructions. The instructions for the instruction kernel may be carried out asynchronously, and the instruction kernel may use a predefined wait time period, a wait state of the data bus, an interrupt, or other means, to signal the end of such an instruction execution.

General Decoder

[0076] As described earlier, the general decoder*110* has a carry number input, a start address input, an end address input, all of which from the input/output control unit *103*, and a plurality of element control bit outputs *111*, each of which connecting exclusively to the enable bit input of a unique memory element *108*. The element address of each memory element *108* is actually decided by the general decoder *110*.

[0077] Inside the general decoder*110*, the carry number input is connected to a carry pattern generator, which positively asserts all its bit outputs whose addresses are an incre-

ment of the inputted carry number while negatively asserting all the other bit outputs. All possible values of the carry number form a set C. A bit output D has an address A, whose binary expression is C(A), and whose natural number factors forming another set Q(A). K(A) is the overlap set between C and Q(A). Using K(A)[k] to denote a unique element of K(A), the logic expression of D[A] is:

[0078] $D[0] = 1;$

[0079] $IF\ A \in K[A]: D[A] = \Sigma_k D[K(A)[k]] + C(A);$

[0080] $ELSE: D[A] = \Sigma_k D[K(A)[k]];$

[0081] The above expression is transformed into standard product-of-sum format using either K-map or Quine-McCluskey method, and the carry pattern generator is constructed using corresponding two-level gates. The product-of-sum construct is chosen for expansibility, so that the addition of C[M] input bit appends !C[M] product term to the existing expressions of (C[M-1] ... C[0]). For an example, a 3/8 carry pattern generator inputs binary carry number (C[2] C[1] C[0]), and outputs bit outputs (D[7] D[6] D[5] D[4] D[3] D[2] D[1] D[0]) in the following manner:

[0082] $D[0] = 1;$

[0083] $D[1] = !C[2]\ !C[1]\ C[0];$

[0084] D[2] = !C[2] C[1] !C[0] + D[1];

[0085] D[3] = !C[2] C[1] C[0] + D[1];

[0086] D[4] = C[2] !C[1] !C[0] + D[2] + D[1];

[0087] D[5] = C[2] !C[1] C[0] + D[1];

[0088] D[6] = C[2] C[1] !C[0] + D[3] + D[2] + D[1];

[0089] D[7] = C[2] C[1] C[0] + D[1];

[0090] Or:

[0091] D[0] = 1;

[0092] D[1] = !C[2] !C[1] C[0];

[0093] D[2] = !C[2] (C[1] + C[0])(!C[1] + !C[0]);

[0094] D[3] = !C[2] C[0];

[0095] D[4] = (C[2] + C[1] + C[0])(!C[2] + !C[1])(!C[1] + !C[0])(!C[2] + !C[0]);

[0096] D[5] = !C[1] C[0];

[0097] D[6] = (!C[2] + !C[0])(C[1] + C[0]);

[0098] D[7] = (!C[2] + C[1])(C[2] + !C[1]) C[0];

[0099] The bit outputs of the carry pattern generator D = (D[N-1] ... D[0]) are connected to the bit inputs of a parallel left

shifter, whose shift amount input S = (S[M-1] ... S[0]) is connected from the start address input to the general decoder *110*. The parallel left shifter concurrently shifts all bit inputs D = (D[N-1] ... D[0]) toward higher address by the amount of shift amount input S at its bit outputs H = (H[N-1] ... H[0]), mathematically as:

[0100] IF A => S: H[A] = D[A - S];

[0101] ELSE: H[A] = 0;

[0102] Since shifting is accumulative, each S[j] input bit just shifts each of all the inputs by the amount of 2^j toward higher address. For an example, the circuit diagram of a 3/8 parallel left shifter is shown in FIG. 4, in which (D[7] ... D[1] D[0]) is the 8-bit input, (H[7] ... H[1] H[0]) is the 8-bit output, and (S[2] S[1] S[0]) is the 3-bit shift amount input. The circuit diagram is readily to be extended when the bit count of inputs and outputs is more than 8.

[0103] Inside the general decoder*110*, the end address input is connected to the address input E = (E[M-1] ... E[0]) of an all-line decoder, which activates all its bit outputs F = (F[N-1] ... F[0]) whose address is less than or equal to the input address. For an example, a 3/8 all-line decoder inputs 3-bit address (E[2] E[1] E[0]), and outputs 8-bit bit

outputs (F[7] ... F[1] F[0]) in the following manner:

[0104] F[7] = E[2] E[1] E[0];

[0105] F[6] = E[2] E[1] !E[0] + F[7];

[0106] F[5] = E[2] !E[1] E[0] + F[6];

[0107] F[4] = E[2] !E[1] !E[0] + F[5];

[0108] F[3] = !E[2] E[1] E[0] + F[4];

[0109] F[2] = !E[2] E[1] !E[0] + F[3];

[0110] F[1] = !E[2] !E[1] E[0] + F[2];

[0111] F[0] = !E[2] !E[1] !E[0] + F[1];

[0112] Or:

[0113] F[7] = E[2] (E[1] E[0]);

[0114] F[6] = E[2] (E[1]);

[0115] F[5] = E[2] (E[1] + E[0]);

[0116] F[4] = E[2] 1;

[0117] F[3] = E[2] + (E[1] E[0]);

[0118] F[2] = E[2] + (E[1]);

[0119] F[1] = E[2] + (E[1] + E[0]);

[0120] F[0] = E[2] + 1;

[0121] The corresponding circuit diagram is displayed in FIG. 5. Assuming the bit output is F[E, N], in which N denotes the bit width of the address input and E denotes the address of the bit output, an all-line-decoder with input address bit width of (N + 1) can be built from an all-line-decoder with input address bit width of N using the logic expression of F[E, N]:

[0122] F[0, 1] = 1;

[0123] F[1, 1] = E[0];

[0124] F[(0 E[N-1] ... E[0]), N+1] = F[(E[N-1] ... E[0]), N] + E[N];

[0125] F[(1 E[N-1] ... E[0]), N+1] = F[(E[N-1] ... E[0]), N] E[N];

[0126] Inside the general decoder *110*, the bit outputs of the parallel left shifter H = (H[N-1] ... H[0]) are AND-combined with the corresponding bit outputs of the all-line decoder F = (F[N-1] ... F[0]), to become the corresponding bit outputs of the general decoder *110*, as illustrated in FIG. 6. All the element control bit outputs are activated *123* whose element addresses are: (A) no less than a start address *121*, (B) no more than an end address *122*, and (C) an integer increment of the carry number starting from the start

address *120*.

[0127] As described later, the value of the carry number input needs not exceed the square root of the total bit output count of the general decoder.

[0128] If the carry number is a constant of 1, the start address is input into a first all-line decoder whose outputs are negatively assertive, and the end address is input into a second all-line decoder whose outputs are positively assertive. The corresponding outputs from the two all-line decoders are AND-combined, before becoming the bit outputs of the general decoder *110*. This special case of general decoder is called a range decoder.

[0129] Due to the design of the general decoder *110*, changing its start address input may be less efficient than changing its end address input in terms of the number of gates that need to change their states.

[0130] It is possible to enable each memory element by the bit storage cell only (without using the general decoder), like conventional processor array. Other means then is used to setting the values of the bit storage cell serially, such as using a controlling CPU. However, this method may be slow for task switching between different array or different members of array items of a same array. Thus, general

decoder or range decoder also may be very useful in controlling processor array in general.

CONTENT MOVABLE MEMORY

[0131] The simplest CP memory is a content movable memory. FIG. 7 shows its memory element *108*. Each memory element *108* has only one addressable register *106*, thus the element address is same as the register address of the addressable register *106*. Through neighborhood connection *114*, the addressable register *106* is also the neighboring register. The memory element has another register, the operation register *200*, which is made of cheap dynamic memory cells that only need to keep their values for more than one clock cycles. A multiplexer *212* selects a neighborhood connection, either (A) from the memory element which has immediately lower element address *114a* or (B) from the memory element which has immediately higher element address *114b*, to copy to the operation register *200* when the write control bit *244* of the operation register *200* is positively asserted. The value of the operation register *200* can be copied to the addressable register *106* when the write control bit *243* of the addressable register *106* is positively asserted. The concurrent bus *109* has two bits, one *241* to select the source of the multiplexer

212 from either *114a* or *114b*, the other *242* to select copying to one of the two registers, *200* or *106*. The enable bit input*111* is AND combined with the other bit *242* of the concurrent bus *109*, to disable any copying when the enable bit input *111* is negatively asserted. Thus, the control unit of the memory element *108* comprises the connections of the multiplexer *212*, the AND gate for the write control bit *243* of the addressable register *106*, the AND gate for the write control bit *244* of the operation register *200*, and the enable bit input*111*.

[0132] The content of addressable registers *106* in the neighboring memory elements can be copied to the addressable register *106* by first being copied through the neighborhood connection *114a* or *114b* to the operation register *200*, and then to the addressable register *106* of the memory elements.

[0133] A content movable memory needs neither priority encoder nor parallel counter *113*. A range decoder is used as the general decoder*110*, so that all the memory elements are activated if their element address is: (A) no less than a start address, and (B) no more than an end address. In this way, the data within a register address range can be moved within a content movable memory.

[0134] Using the contenting moving procedure, a content movable memory can add, remove, relocate, and change size of a stored data object anywhere within it while keep its content closely packed. It may contain a truly dynamic array without the need for either link list or look-ahead allocation. It may even use address independent unique ID to identify each stored data objects, and support containment relationship so that: (A) when the size of a contained data object is changed, the container data object is changed accordingly, and (B) when the container data object is removed, all the contained data objects are removed.

[0135] When using a content movable memory for a program, the space allocated for a variable can grow and shrink easily according to the need, which brings about the following advantages: (1) a numerical variable will never go out of range, (2) an array is always dynamic; (3) the distinction between stack memory and heap memory may no longer need, and (4) the most economical use of the resources can be achieved.

[0136] Since both size and precision for each numerical variable is adjustable dynamically, the conventional float fractional formats and their rules of operations can be improved so

that the precision error is always limited to the LSB (least significant bit) of the mantissa. For an example, the result precision of an addition or subtraction is the lesser precision of the two operands, and in case the two operands having same precision, the result precision remains in the original LSB if the two operands are independent from each other, and the operation on the original LSBs does generate carry, or it is shifted to the bit immediately above the original LSB if otherwise. The multiplication, division, and other arithmetic operations can be based upon similar rules for addition and subtraction. In such a scheme, each numerical value is guaranteed to be precise until LSB. In worst case, instead of giving wrong answer due to precision error accumulation and propagation as in the conventional float fractional math, the new float fractional math may indicate that at a certain step of the algorithm, the initial values are no longer precise enough for the algorithm.

CONTENT MATCHABLE MEMORIES

[0137] Content matchable memory is also a family name. It has three types of memory element:

[0138] (1) content searchable memory element, which can match the content of its addressable register *106*with a datum,

and positively assert its match bit output *112* if (I) its enable bit input *111* is positively asserted, and (II) the comparison satisfies the match requirement, which can be any of: (A) equal, and (B) unequal. Neighborhood connection allows comparison between a datum and the collective content of any neighboring memory elements. Thus, the primary use is to find all matching strings among a text.

[0139] (2) content comparable memory element, which can compare the content of its addressable register *106*with a datum, and positively assert its match bit output *112*if (I) its enable bit input *111* is positively asserted, and (II) the comparison satisfies the match requirement, which can be any of: (A) equal, (B) unequal, (C) larger, (D) smaller, (E) larger or equal, and (F) smaller or equal. Neighborhood connection allows comparison between a datum and the collective content of neighboring memory elements which forms the items of an array. Thus, the primary use is to find all matching array items.

[0140] (3) It is also possible to combine either a content searchable memory element or a content comparable memory element with a content movable memory element.

[0141] FIG. 8a shows a content searchable memory element *108*. It has only one addressable register *106*, whose content is

to be searched. The concurrent bus *109* sends: (A) a mask *204*, which is AND combined with the addressable register *106* at a bus AND gate *261*; (B) the datum to be matched *205*, whose value is compared with the masked data from the output of the AND gate *261* at a comparator *211*, which composed of a bus XOR gate and a OR gate; and (C) the instruction *207*, which contains the requirement of matching. The mask *204* of the concurrent bus *109*, and the AND gate *261* are optional, and the addressable register *106* may be compared directly with the datum to be matched *205* of the concurrent bus *109* at the comparator *211*. The bit output of the comparator is positively asserted if the masked datum at the addressable register *106* differs from the datum to be matched *205* at any bit, which is the "case" of the comparison. The instruction *207* portion of the concurrent bus *109* contains a "condition" code bit *252*, which is compared with the "case" of the comparison at a XOR gate *260*, whose bit output is positively asserted if the "case" does not equals the "code". The bit output from the XOR gate *260* is AND combined with the enable bit input *111* at an AND gate *262* whose output asserts the match bit output *112* of the memory element *108*.

[0142] Additional logic allows the value matching across memory

elements when neighboring elements to be matched together. Instead of directly connecting to the AND gate *262*, the bit output from the XOR gate *260*is connected to an AND gate *263*, to be saved into a one-bit neighboring register *201*, whose write control bit is connected to the enable bit input *111*of the memory element *108*, and whose bit output is connected to the AND gate *262*which drives the match bit output *112* of the memory element *108*. The one-bit neighboring register *201* is connected to the neighboring memory elements through neighborhood connection *114*. The concurrent bus *109* sends one more instruction bit "self" *253* with the instruction portion *207* of the concurrent bus *109*. Through an OR gate *264*, when the instruction bit "self" *253* is positively asserted, the match bit output *112* is positively asserted when a match is found by the XOR gate*260*; otherwise, the neighborhood connection from the memory element whose element address is higher *114b*also has to be positively asserted to positively assert the match bit output *112*. Assuming the width of the addressable register *106* of each of all the memory elements is byte, an algorithm for a search of a string is the following:

[0143] (1) Match for equal the addressable register *106* with the

highest byte of the value, while positively asserting the instruction bit "self" *253*;

[0144] (2) In the order from high to low, match for equal the addressable register *106* with the corresponding byte of the value, while negatively asserting the instruction bit "self" *253*;

[0145] (3) The memory elements whose match bit outputs are positively asserted are the memory elements which have the smallest element addresses of neighboring memory elements which hold the string to be searched.

[0146] Similar construct can be built for the algorithm to match for equal in the order from low to high, or from both directions.

[0147] FIG. 8b shows a content comparable memory element *108*. It has only one addressable register *106*, whose content is to be compared. The concurrent bus *109* sends: (A) a mask *204*, which is AND combined with the addressable register *106* at a bus AND gate *261*; (B) the datum to be compared *205*, whose value is compared with the masked datum from the output of the AND gate *261* at a comparator *211*; and (C) the instruction *207*, which contains the requirement of comparison. The mask *204* of the concurrent bus *109*, and the AND gate *261* are optional, and the address-

able register *106* may be compared directly with the datum to be compared *205* of the concurrent bus *109* at the comparator *211*. The "=" and ">" outputs of the comparator *211* is the "case" of comparing the masked value of the addressable register *106* and the datum to be compared *205*, while the first three bits *250* to *252* of the instruction *207* portion of the concurrent bus *109* contains the "condition" code of the match requirements. A matching logic table*260* of standard two-layer logic combines the "case" and the "condition", to positively assert its output if the "case" matches the "condition", as demonstrated by the following function table for the match output from the matching logic table*260*:

Function of matching logic table

	Cond	000	001	01X	11X	100	101
Case	Mean	<	>	!=	==	<=	>=
00	<	1	0	1	0	1	0
01	>	0	1	1	0	0	1
1X	==	0	0	0	1	1	1

[0148] The bit output from the matching logic table *260* is AND combined with the enable bit input *111*at an AND gate *262* whose output asserts the match bit output *112* of the memory element *108*

[0149] Additional logic allows the value matching across memory elements when each of the items to be matched spans several neighboring elements. Instead of directly connecting to the AND gate *262*, the bit output from the matching logic table *260*is connected to an AND gate *263*, to be saved into a one-bit neighboring register *201*, whose write control bit is connected from the enable bit input *111* of the memory element *108*, and whose bit output is connected to the AND gate *262* which drives the match bit output *112* of the memory element *108*. The one-bit neighboring register *201* is connected to the neighboring memory elements through neighborhood connection *114*. The concurrent bus *109* sends three more instruction bits: "self" *253*, "transfer" *254*, and "select" *255*, with the instruction *207* portion of the concurrent bus *109*. When the instruction bit "select" *255* is positively asserted, the neighborhood connection from the memory element whose element address is immediately higher *114b* is selected to the output of a multiplexer *265*; otherwise, the neighborhood connection from the memory element whose element address is immediately lower *114a* is selected. Through an OR gate *264*, when the instruction bit "self" *253* is positively asserted, the output of the AND

gate *263* is positively asserted when a matched is found by the matching logic table *260*; otherwise, the output of the multiplexer *265* also has to be positively asserted to positively assert the output of the AND gate *263*. Through a multiplexer *266* and a AND gate *267*, when the instruction bit "transfer" *254* is positively asserted, and the neighboring register *201* is also positively asserted, the output of the multiplexer *265* is saved into the neighboring register *201*; otherwise, the output of the AND gate *263* is saved into the neighboring register *201*.

[0150] For simplicity of discussion: (A) the bit width of the addressable register *106* in each memory element is byte; (B) each item contains M neighboring memory elements, which are denoted as (M-1)th to 0th in the order of from high to low in element address containing (M-1)th to 0th significant bytes of the value of the item; (C) the value to be matched is a M-byte unsigned value; and (D) the action of setting the general decoder *110* accordingly is omitted, which is somewhat obvious.

[0151] An algorithm for an equal matching is the following:

[0152] (1) For all the (M-1)th memory elements of all the items, match for equal the addressable register *106* with the (M-1)th significant byte of the value, while: (A) positively

asserting the instruction bit "self" *253*; and (B) negatively asserting the instruction bit "transfer" *254*. Step (1) positively asserts the neighboring registers *201* of all the (M-1)th memory elements when each of their addressable registers *106* has value equal to the (M-1)th significant byte of the value to be matched.

[0153] (2) Letting j be (M-2), for all the jth memory elements of all the items, match for equal the addressable register *106* with the jth byte of the value, while: (A) negatively asserting the instruction bit "self" *253*; (B) negatively asserting the instruction bit "transfer" *254*; and (C) positively asserting the instruction bit "select" *255*;. Step (2) positively asserts the neighboring registers *201* of each of all the jth memory elements when: (A) the addressable register *106* has value equal to the jth significant byte of the value to be matched, and (B) the neighboring memory element of (j+1)th significance has positively asserted neighboring register *201*.

[0154] (3) Repeat step (2) with j decreased from (M-2) to 0. Step (3) positively asserts the neighboring registers *201* of the consecutive memory elements of each of all the array items whose addressable registers *106* all have values equal to the corresponding bytes of the value to be

matched from highest significance.

[0155] (4) The array items which equal the value to be matched have their neighboring registers *201* of 0th memory elements positively asserted.

[0156] An algorithm to compare the value of all the array items with a value to be matched for a requirement other than (A) equal, or (B) unequal, is the following:

[0157] (1) For all the (M-1)th memory elements of all the items, match for equal the addressable register *106* with the (M-1)th significant byte of the value, while: (A) positively asserting the instruction bit "self" *253*; and (B) negatively asserting the instruction bit "transfer" *254*. Step (1) positively asserts the neighboring registers *201* of all the (M-1)th memory elements when each of their addressable register *106* has value equal to the (M-1)th significant byte of the value to be matched.

[0158] (2) Letting j be (M-2), for all the jth memory elements of all the items, match for equal the addressable register *106* with the jth byte of the value, while: (A) negatively asserting the instruction bit "self" *253*; (B) negatively asserting the instruction bit "transfer" *254*; and (C) positively asserting the instruction bit "select" *255*;. Step (2) positively asserts the neighboring registers *201* of each of all the jth

memory elements when: (A) the addressable register *106* has value equal to the jth significant byte of the value to be matched, and (B) the neighboring memory element of (j+1)th significance has positively asserted neighboring register *201*.

[0159] (3) Repeat step (2) with j decreased from (M-2) to 1. Step (3) positively asserts the neighboring registers *201* of the consecutive memory elements of each of all the array items whose addressable registers *106* all have values equal to the corresponding bytes of the value to be matched from highest significance.

[0160] (4) For all the 0th memory elements of all the items, match for the requirement the addressable register *106* with the 0th significant byte of the value to be matched, while: (A) positively asserting the instruction bit "self" *253*; and (B) negatively asserting the instruction bit "transfer" *254*. Step (4) positively asserts the neighboring registers *201* of the 0th memory elements when the addressable register *106* has value satisfying the match requirement with the 0th significant byte of the value to be matched.

[0161] (5) Letting j be 1, for all the jth memory elements of all the items, match for the requirement the addressable register *106* with the jth byte of the value to be matched,

while: (A) positively asserting the instruction bit "self" *253*; (B) positively asserting the instruction bit "transfer" *254*; and (C) negatively asserting the instruction bit "select" *255*. When a neighboring register *201* is originally positively asserted, it is filled with the value of the neighboring register *201* from the neighboring memory element of (j-1)th significance; otherwise, it is positively asserted when the addressable register *106* has value satisfying the match requirement with the jth significant byte of the value to be matched.

[0162] (6) Repeat Step (5) with j increased from 1 to (M-1). At last, the match bit outputs *112* from the (M-1)th memory elements of all the items are positively asserted when the array item which is held by neighboring memory elements matches the value to be matched according to the requirement.

[0163] The above algorithm can be extended easily to array each item of which contains memory elements whose addressable registers *106* have width other than byte, or whose content significance is in reverse order with the element address, or matching signed values.

[0164] Instead of comparing the value of a register and a value to be matched on the concurrent bus *109*, it is also possible

the matching is between two addressable registers *106* within each memory elements.

[0165] When the content of the enabled memory elements are distinguished, content matchable memory has three ways to collect the positively asserted match bit outputs *112* using:

[0166] (1) a priority encoder *113* to find either the highest or the lowest element address of the match bit outputs *112* which have been positively asserted.

[0167] (2) a parallel counter *113* to count the match bit outputs *112*which have been positively asserted.

[0168] (3) the combination of (1) and (2).

[0169] An algorithm for enumerating matched array items is:

[0170] (1) Assert positively the match bit outputs *112* of all the matched items concurrently.

[0171] (2) Set the priority of the priority encoder *113* to be from high to low.

[0172] (3) If the no-hit bit output of the priority encoder *113*is positively asserted, all the matched items have been enumerated, and the enumerating algorithm should be terminated.

[0173] (4) Read the address output of the priority encoder *113*,

which contains the highest element address of the matched item between the start address and the end address.

[0174] (5) Set the end address to the item whose element address is immediately lower than that of the item which has been found in step (4).

[0175] (6) Repeat step (3) to step (5).

[0176] It is easy to design an alternative algorithm similar to the above algorithm based on the low-to-high priority of the priority encoder *113*. Due to the design of the general decoder*110*, changing its start address input may be less efficient than changing its end address input.

[0177] An algorithm for counting matched array items is:

[0178] (1) Assert positively the match bit outputs *112* of all the matched items concurrently.

[0179] (2) Read the count output of the parallel counter *113*, which contains the count of the matched memory elements.

[0180] An algorithm to construct a histogram of M sections is:

[0181] (1) Designate a variable CNT_HIGH.

[0182] (2) Designate a variable CNT_LOW.

[0183] (3) Match for smaller all the items with the upper limit of the smallest section.

[0184] (4) Read the count output of the parallel counter *113* into CNT_LOW, which contains the histogram count of the smallest section.

[0185] (5) Let j be 1, match for smaller all the items with the upper limit of the jth section.

[0186] (6) Read the count output of the parallel counter *113* into CNT_HIGH.

[0187] (7) Subtracting CNT_LOW from CNT_HIGH to obtain the histogram count of the jth section.

[0188] (8) Copy CNT_LOW from CNT_HIGH.

[0189] (9) Repeat Step (5) to (8) for j from 2 to (M-1).

[0190] (10) Subtracting CNT_HIGH from the total count of the items to obtain the histogram count of the largest section.

[0191] The histogram of the data can be used to estimate the sum and the distribution of the data.

[0192] It is possible that the concurrent bus *109* sends no instruction *207*, and the matching is done in a predefined manner, such as (A) always searching for equal between the content of the addressable register *106* of each of all the enabled memory elements and the condition datum

205 of the concurrent bus *109*, or (B) always searching for equal between the contents of two addressable registers *106* of each of all the enabled memory elements. The usefulness of such arrangement is limited.

PARALLEL COMPARATOR

[0193] To facilitate quick value comparison, a parallel comparator may be used as the comparator *211* in the memory elements *108*. An example of 4-bit parallel comparator is shown in FIG. 9. A parallel comparator inputs two numbers, X = (X[2^N-1] ... X[0]), and Y = (Y[2^N-1] ... Y[0]), in which X[j] and Y[j] denote the jth significant bit of the input numbers of bit width 2^N. When X and Y are equal, the parallel comparator positively asserts its equal bit output "X = Y". Otherwise, it positively asserts its larger bit output "X > Y" when X is larger than Y, or negatively asserts the larger bit output "X > Y" when X is smaller than Y, and outputs the largest bit significance of the X and Y difference at its address output A = (A[N-1] ... A[0]), in which A[j] denotes the jth significant bit of the address A of bit width N. In the first step, each pair of X[j] and Y[j] are compared to obtain G[j] and L[j], which are positively asserted when X[j] > Y[j] and X[j] < Y[j] respectively, as:

[0194] G[j] = X[j] !Y[j];

[0195] L[j] = !X[j] Y[j];

[0196] In the second step, the corresponding bits of G and L are OR-combined to obtain the exclusive-OR combination Z[j] of X[j] and Y[j], as:

[0197] Z[j] = G[j] + L[j];

[0198] In the third step, each of all the bits of Z is connected to the input bit of an encoder *271* of high-to-low priority with the bit's significance in Z being the same as the input bit's address of the encoder *271*, the address at the address output (A[N-1] ... A[0]) of the encoder *271* thus contains the most significance of the bit at where X and Y differs, and the no-hit bit output of the encoder *271*, which is the equal bit output "X = Y" of the parallel comparator, is positively asserted when X and Y are equal.

[0199] In the forth step, the address output of the encoder is connected to the address input of a multiplexer *272*. Each of all the bits of G is connected to the input bit of the multiplexer with the bit's significance in G being the same as the input bit's address, so that the bit output of the multiplexer *272*, which is the larger bit output "X > Y" of the parallel comparator, is positively asserted when X is

larger than Y, or negatively asserted when X is smaller than Y.

PARALLEL ADDER

[0200] A parallel adder adds two numbers X and Y into a number S in two steps:

[0201] (1) Adds all corresponding bits of X and Y simultaneously without considering carrying over from other bits. Let n denote the nth bit, Z denote the bitwise XOR combination of X and Y, and C denote the carry number:

[0202] Z[n] = X[n] XOR Y[n] = (X[n] + Y[n]) !(X[n] Y[n]);

[0203] C[n] = X[n-1] Y[n-1], with C[0] as carry input;

[0204] IF Z[n-1] = 1 THEN C[n] = 0; IF C[n] = 1 THEN Z[n-1] = 0;

[0205] (2) Adds the Z and C into S. Let "1...1" denotes a continuous 1 of bits of any length, and let "?" denotes an unknown value, the general cases for adding any fragment of Z and C bits is:

Parallel addition cases

Case	I	II	III	IV
Z	0	00...0	01...10	01...10
C	0	1...10	0...01?	0...00?
S	?	1...1?	10...0?	01...1?

[0206] Case I and II show that whenever Z[n] is 0, there is no carry over beyond this bit. Case III and IV shows how carry is generated. The general equations for the sum S are:

[0207] $A[n, j] = C[n-j] \Pi_{k=1 \text{ to } j} Z[n-k];$

[0208] $A[n] = \Sigma_{j=1 \text{ to } n} A[n, j];$

[0209] $S[n] = !Z[n]\ C[n] + Z[n]\ !C[n]\ !A[n] + !Z[n]\ A[n];$

[0210] The equation of A[n] defines the carry look-ahead logic of the parallel adder, which can be implemented by an OR gate which adds the outputs from a series AND gate, each of which implements an A[n, j] of a different j. Due to large number of inputs, simplified AND and OR gate symbols are used, which are commonly used for transmission gate logic. FIG. 10 shows the examples of the standard and simplified three-input AND gate symbol. FIG. 11 shows an example of a 4-bit parallel adder.

[0211] A by-product of the above parallel adder implementation is the outputs for the bitwise AND, OR, and XOR outputs of X and Y.

PARALLEL COUNTER

[0212] As described earlier, either a priority encoder or a parallel counter or both may be used in concurrent matching operations. A priority encoder is a standard device. A parallel

counter concurrently counts the bit inputs which are positively asserted simultaneously and outputs the count at its output. An N-bit parallel counter has 2^N bit inputs and N-bit output.

[0213] A parallel counter can be constructed using parallel adders in binary tree construct, in which each parallel adder counts two inputs to its output at each tree node. The binary tree construct is made of layers of notes of same parallel adders. The jth layer contains 2^(N-j) j-bit parallel adders, each of which adds two j-bit inputs X = (X[j] ... X[0]) and Y = (Y[j] ... Y[0]) from the previous layer, and outputs (j+1)-bit output S = (S[j+1] S[j] ... S[0]) to the next layer. FIG. 12 shows the binary tree construct of a 4-bit parallel counter comprising 16 bit inputs, a 1st layer *151* of 8 1-bit parallel adders, a 2nd layer *152* of 4 2-bit parallel adders, a 3rd layer *153* of 2 3-bit parallel adders and a 4th layer *154* of 1 4-bit parallel adders.

[0214] In the following tables, the item at the first column of the first row marks the layer number, the first row contains the values for X, the first column contains the values for Y, and the rest items contains the corresponding bit output of the parallel adder:

1st layer 1-bit adder

(1)	0	1
0	00	01
1	01	10

2nd layer 2-bit adder

(2)	00	01	10
00	000	001	010
01	001	010	011
10	010	011	100

3rd layer 3-bit adder

(3)	000	001	010	011	100
000	0000	0001	0010	0011	0100
001	0001	0010	0011	0100	0101
010	0010	0011	0100	0101	0110
011	0011	0100	0101	0110	0111
100	0100	0101	0110	0111	1000

[0215] As a general rule, when the (j+1)th bit output is positively asserted for a j-bit parallel adder in a jth layer, its other bit outputs are negatively asserted. There is also no carry bit input. Thus, the parallel adders on each node of the binary tree of the parallel counter can be simplified accordingly by: (A) removing the carry look-ahead logic for the most significant bit; and (B) starting the carry look-ahead logic from the 1st bit. FIG. 13 shows such a 4-bit

parallel counter.

[0216] An alternative way for constructing a small-scale parallel counter of high speed is to: (1) use resistors to convert logic inputs into currents, (2) use GHz op-amp to add these currents together and convert the current sum to voltage, then (3) use GHz D/A converter to convert the voltage to binary number. A 3-bit parallel counter of such construct is shown in FIG. 14. In the first stage, the currents of 7 bit inputs (D6 D5 D4 D3 D2 D1 D0) driving 7 resisters of identical resistance R are summed up by the first op-amp *131*, which has a feedback resistor of $1/4\,R$. When the count of the positively asserted bit input is equal to or larger than 4, the voltage at the output of the first op-amp *131* is equal or larger than the voltage of logic 1, thus the output C2 of the first analog comparator *132* is positively asserted, which is the most significant bit of the counter output; otherwise, C2 is negatively asserted. Through analog switches *133* and *135*, when C2 is positively asserts, a voltage of logic 1 is subtracted from the output of the first op-amp *131* by a second op-amp *134*; otherwise, the output of the first op-amp *131* is passed directly to the next stage. The input at the next stage is scaled up by 2-fold by the third op-amp *136*, to

find the bit C1 of the counter output by a second analog comparator *137*. Same procedure goes on until all bits of the counter output are found. In this way, a fast parallel counter is constructed with fairly small number of op-amps. Such scheme can be extended to 255-inputs and 8-bit outputs using 16 op-amps, 16 analog switches and 8 analog comparators.

[0217] A (2N)-bit parallel counter of slightly slower speed can be made of three layers of N-bit parallel counters. An example of constructing a 6-bit parallel counter using 3-bit parallel counters is shown in FIG. 15. The first layer *141* is consisted of (2^N + 1) N-bit parallel counters counting (2^(2N) - 1) bit inputs. Out of them, the corresponding digit of the counter outputs of (2^N - 1) smaller parallel counters are counted by N smaller parallel counters in the second layer *142*. For an example, a second-layer N-bit counter *144*counts the 1st bit outputs of the first layer N-bit counters. Except their most significant bits, the counter outputs of the rest two smaller parallel counters in the first layer are counted by an additional N-bit parallel counter *145* in the second layer. The outputs from the 2nd layer N-bit counters *142* are added together by several smaller parallel counters connected as ripple 1-bit

adders in the 3rd layer *143*, each of them functions like a multiple inputs and multiple carry outputs 1-bit adder. For an example, a third-layer N-bit counter *146* is connected as a multiple carry-in and multiple carry-out 1-bit adder for the 2nd bit output of the (2N)-bit counter. A conventional 1-bit adder *147* may be used for the 0th bit output of the (2N)-bit counter. Using this technique, a 16-bit output parallel counter of 6-cycle delay can be made of two hundred and sixty-eight 8-bit output parallel counters, and one 6-bit output parallel counter.

MULTI-CHANNEL MULTIPLEXER AND DEMULTIPLEXER

[0218] A multi-channel multiplexer selects a channel width number of consecutive bit inputs starting from a bit address. When the channel width is non-zero, the bit address not only selects the corresponding bit input to the LSB output, but also the bit input which has immediately higher bit address to the next-to-LSB output, and so forth. A multi-channel demultiplexer is the functionally reverse of the corresponding multi-channel multiplexer. An example of 8-input 4-channel multiplexer is shown in FIG. 16. The channel inputs are (X7 X6 X5 X4 X3 X2 X1 X0). The Channel outputs are (Z3 Z2 Z1 Z0). The channel width selections are (W1 W0). The channel address inputs are (A2 A1

A0). (A2 A1 A0) selects one of (X7 X6 X5 X4 X3 X2 X1 X0) as Z0 in the same manner as a normal multiplexer. When either W1 or W0 is positively asserted, (A2 A1 A0) selects one of (X7 X6 X5 X4 X3 X2 X1) as Z1, which has immediately higher input bit address than Z0. When W1 is positively asserted, (A2 A1 A0) selects one of (X7 X6 X5 X4 X3 X2) as Z2, which has immediately higher input bit address than Z1. When both W1 and W0 are positively asserted, (A2 A1 A0) selects one of (X7 X6 X5 X4 X3) as Z3, which has immediately higher input bit address than Z2. Thus, the number of valid bit outputs is determined by the value of the channel width selections (W1 W0). The corresponding 8-output 4-channel demultiplexer is shown in FIG. 17. The channel inputs are (X3 X2 X1 X0). The Channel outputs are (Z7 Z6 Z5 Z4 Z3 Z2 Z1 Z0). The channel width selections are (W1 W0). The channel address inputs are (A2 A1 A0). (A2 A1 A0) selects one of (Z7 Z6 Z5 Z4 Z3 Z2 Z1 Z0) from X0 in the same manner as a normal demultiplexer. When either W1 or W0 is positively asserted, (A2 A1 A0) selects one of (Z7 Z6 Z5 Z4 Z3 Z2 Z1) from X1, which has immediately higher input bit address than X0. When W1 is positively asserted, (A2 A1 A0) selects one of (Z7 Z6 Z5 Z4 Z3 Z2) from X2, which has immediately

higher input bit address than X1. When both W1 and W0 are positively asserted, (A2 A1 A0) selects one of (Z7 Z6 Z5 Z4 Z3) from X3, which has immediately higher input bit address than X2. Thus, the number of valid output channels is determined by the value of the channel width selections (W1 W0).

Construct of a Database/Math Memory Element

[0219] The memory elements are the basic units within a CP memory that store and process data, each of which comprises preferably at least one addressable registers, possibly other registers, a control unit, and some processing power. FIG. 18 shows the memory element construct of a math memory, which could be either math 1D memory or math 2D memory, which only differs in number of neighborhood connections. It can be turned into the memory element of a database memory by deleting components from it, as described later in this Description.

[0220] Most conventional massive parallel architectures implore bit serial operation to save semiconductor construct on each processing element. The CP memory may use some new hardware components such as parallel comparator and multi-channel multiplexer and multi-channel demultiplexer, or improved hardware component such as paral-

lel adder, to implore bit parallel operation to improve the performance without paying a high price in semiconductor construct for each processing element.

[0221] The registers within memory element can be categorized as either addressable register or internal register, depending on whether it is accessible by the exclusive bus *105* and thus from outside the CP memory using the register address of the register. All the registers in FIG. 18 are addressable registers. In this way, while the CP memory is concurrently processing one set of registers, the other set of registers can be prepared for another task by exclusive access means such as direct memory access, since the exclusive bus *105* and the concurrent bus *109* within a CP memory can work independently from each other.

[0222] Some registers have special functions.

[0223] (1) One register of the memory element is a neighboring register *201*, which is connected concurrently to neighboring memory elements through neighborhood connections *114*. Such connections from different two neighboring memory elements are *114a* and *114b* respectively for a math 1D memory or a database memory. A math 2D memory has four such connections from different four

neighboring memory elements in each of its memory elements. Except the neighboring memory element count and the partition of element address into X address and Y address, a math 2D memory is otherwise identical to a math 1D memory.

[0224] (2) One register of the memory element is a status register *203*. When being activated by the exclusive connection *111* to the enable bit input of the control unit *210*, a memory element can have internal states, which is determined by inputs to the control unit *210*. Some of the bits of a status register *203* are connected to the control unit *210* through connection *209*, and can be set or reset by the control unit *210*. The status register *203* contains a carry bit and at least one status bit.

[0225] (3) One register of the memory element is an operation register *200*. A bit multiplexer/demultiplexer *213*, which is a multi-channel multiplexer/demultiplexer, can either selectively read any bit section of the operation register *200* when the write control bit *226* is negatively asserted, or selectively write any bit section of the operation register *200* when the write control bit *226* is positively asserted.

[0226] (4) The rest registers *202* of the memory element are data registers. A register multiplexer/demultiplexer *212*, which

is also a multi-channel multiplexer/demultiplexer, can: either (A) selectively read any bit section of the data registers *202* and the neighboring register *201* of the memory element, the neighboring registers in the neighboring memory elements through the neighborhood connections *114a*, *114b*, etc, and the data portion *204* from the concurrent bus *109* when the write control bit *225* is negatively asserted, or (B) selectively write any bit section of the data registers *202* and the neighboring register *201* of the memory element when the write control bit *225* is positively asserted.

[0227] The concurrent bus *109* carries element instruction to the memory elements in the format of:

[0228] "condition: operation width [bit] register[bit]"

[0229] The bit width of the operant is the "width" code. The value starts from 0 for bit-serial operation, and ends at one less than the bit width of the operation register *200*. It is sent to both the register multiplexer/demultiplexer *212* and the bit multiplexer/demultiplexer *213*. as the channel width inputs

[0230] One operant is the first "[bit]" code, which is a portion *206* of the concurrent bus *109* that is sent to the bit multiplexer/demultiplexer *213* as the address input. When the

write control bit *226* of the bit multiplexer/demultiplexer *213* is negatively asserted, a bit section of the operation register *200* of "width" width starting from bit significance "[bit]" and up is cached at the "read" output *221* of the bit multiplexer/demultiplexer *213* and is denoted as "[bit]" *221*.

[0231] The other operant is the "register[bit]" code, which is another portion *205* of the concurrent bus *109* that is sent to the register multiplexer/demultiplexer *212* as the address input. The "register" could be any one of: its own neighboring register *201* and data registers *202*, its neighbor's neighboring registers *114a*, *114b*, etc, and the data portion *204* on the concurrent bus *109*. The "[bit]" specifies the lowest bit significance of the bit section of "width" width. When the write control bit *225* is negatively asserted, the bit section specified by "register[bit]" is cached at the "read" output *220* of the register multiplexer/demultiplexer *212* and is denoted as "register[bit]" *220*. The data registers *202* may form a random access memory of bits so that a selection of bit section may across register boundary.

[0232] The "condition: operation" portion *207* of the concurrent bus *109* is input into the control unit *210*. The "condition"

code is the condition for finishing executing the "operation width [bit] register[bit]" portion of the instruction. It is implemented by the inputs into the control unit *210* comprising the connection from the status register *209*, the AND- or OR- logic combination *222* of all the bits of "register[bit]" *220* or "[bit]" *221*, and the outputs of a comparator *211* which compares the values of the "[bit]" *221* and the "register[bit]" *220*

[0233] The "condition" code of the instruction can be: (A) none, (B) any one of, (C) the AND or OR combination of any ones from any two categories of:

[0234] (1) "ANY register[bit]", "ALL register[bit]": If any or all the "register[bit]" *220* bits are positively asserted respectively.

[0235] (2) "ANY [bit]", "ALL [bit]": If any or all the "[bit]" *221* bits is positively asserted respectively.

[0236] (3) <, <=, =, !=, >=, >: If the corresponding value relation between the "register[bit]" *220* and the "[bit]" *221* is satisfied.

[0237] (4) R, S: if the status bit of the status register *203* is being negatively or positively asserted respectively.

[0238] (5) E, C: if the carry bit of the status register *203* is being negatively or positively asserted respectively. A database memory has no carry bit in the status register *203*, thus no

this category of "condition" code.

[0239] If the condition is not met, the instruction execution terminates before executing the "operation" code, as if the memory element is not activated.

[0240] The "operation" code is different for a database memory and a math memory.

[0241] The memory elements of a database memory have neither carry bit in its status register *203*, nor adder *214*, nor operation multiplexer *215*, nor op-code outputs *208* of the control units *210*. The "register[bit]"*220* is connected directly to the operation result *222*. Thus, the set of "operation" code contains at least:

[0242] (1) WA (Write address): to positively assert the match bit output *112*.

[0243] (2) WR (Write): to copy the "register[bit]" *220* to the bit section of the operation register *200* specified by "[bit]".

[0244] (3) RD (Read): to copy the "[bit]" *221* to the bit section of any one of its data registers*202* or its own neighboring register *201* specified by "register[bit]".

[0245] (4) CS (Clear Status): negatively assert the status bit of the status register *203*.

[0246] (5) SS (Set Status): positively assert the status bit of the status register *203*.

[0247] The memory element of a math memory is more complex. An adder *214* inputs the "register[bit]" *220*, the "[bit]" *221*, the carry bit of the status register *203*, and outputs the sum to an operation multiplexer *215* while setting the carry bit of the status register *203* accordingly. As by product of adding the "register[bit]" *220* and the "[bit]" *221*, the adder *214* also outputs the bitwise AND-, OR- and XOR- combination of the "[bit]" *221* and "register[bit]" *220* to the operation multiplexer *215*. The operation multiplexer *215* also inputs the "register[bit]" *220*, and the bitwise complement of the "[bit]" *221*. The control unit *210* may select an operation result *222* from an operation multiplexer *215* through an op-code connection *208* and save the operation result *222* to the "[bit]" bit of the operation register *200* by positively asserting the write control bit *226* of the bit multiplexer/demultiplexer *213*. As a result, the set of "operation" code contains at least the addition of:

[0248] (6) NG (Negate): to select the bitwise complement of the "[bit]" *221* as the output *222* of the operation multiplexer *215*, and to copy it to the bit section of the operation register *200* specified by "[bit]". This operation logically inverts each bits of the bit section of the operation register

200 specified by "[bit]".

[0249] (7) ND (AND): to logically AND combine the corresponding bits of the "register[bit]" *220* and the "[bit]" *221*, and to copy the result to the bit section of the operation register *200* specified by "[bit]".

[0250] (8) OR (OR): to logically OR combine the corresponding bits of the "register[bit]" *220* and the"[bit]" *221*, and to copy the result to the bit section of the operation register *200* specified by "[bit]".

[0251] (9) XR (XOR): to logically XOR combine the corresponding bits of the "register[bit]" *220* and the "[bit]" *221*, and to copy the result to the bit section of the operation register *200* specified by "[bit]".

[0252] (10) AD (Add): to add the values of the "register[bit]" *220* and the "[bit]" *221* with the carry bit of the status register *203*, to set the carry bit of the status register *203* from adding, and to copy the result of adding to the bit section of the operation register *200* specified by "[bit]".

[0253] (11) CC (Clear Carry): to negatively assert the carry bit of the status register *203*.

[0254] (12) SC (Set Carry): to positively assert the carry bit of the status register *203*.

[0255] The register multiplexer/demultiplexer *212* and the bit

multiplexer/demultiplexer *213* enable instant bit-wise shift operations of any amount. Thus, each of all the memory elements of a math memory can carry out multiplication and division using a series of addition, subtraction and shift operations. Other math operations are also possible.

[0256] The coding of the element instruction set are designed so that multiple "operation" codes can be carried concurrently by the concurrent bus *109* in a same element instruction for the same "register[bit]" code and the "[bit]" code provided that these "operation" codes may be carried out concurrently without confliction. For an example, the concurrent positively assertion of the write control bit *225* of the register multiplexer/demultiplexer *212* and the write control bit *226* of the bit multiplexer/demultiplexer *213* in the memory elements of a database memory results in exchange the two set of bits of the two registers. Thus, the "operation" codes for "WR" and "RD" should be concurrent for each other.

[0257] All element instruction may have same length and uses one clock cycle, so that the memory element circuit can be treated as combinational logic. The control unit *210* sends pulse signal *231*, *232*, and *233*, to other components of a

database memory element, or pulse signal *231*, *232*, *233*, *234*, and *235* to other components of a math memory element. The timing logic is the following:

[0258] (1) The control unit *210* pulses the enable bit input *231* while negatively asserting the write control bit *225* of the bit & register multiplexer/demultiplexer *212*, to read "register[bit]" bit section to its "read" output *220*. At the same time, the control unit *210* pulses the enable bit input *232* while negatively asserting the write control bit *226* of the bit multiplexer/demultiplexer *213* to read "[bit]" bit section to its "read" output *221*.

[0259] (2a) The control unit *210* pulses the enable bit input *233* of the comparator *211*.

[0260] (2b) At the same time, the control unit *210* of a math memory pulses the enable bit input*234* of the adder *214*.

[0261] (3) If the "condition" code of the instruction is not met, the control unit *210* sends no more timing signals for the instruction cycle, and the instruction execution terminates. Otherwise, the control unit *210* of a math memory pulses the enable bit input *235* of the operation multiplexer *215*.

[0262] (4) According to the "operation" code, the control unit *210* may: (A) pulse the enable bit input *231* while positively as-

serting the write control bit *225* of the register multiplexer/demultiplexer *212*, or (B) pulse the enable bit input *232* while positively asserting the write control bit *226* of the bit multiplexer/demultiplexer *213*, or (C) positively assert the match bit output *112*, or (D) combination of (A) and (B), or (E) combination of (A) and (C), or (F) combination of (B) and (C), or (G) combination of (A) and (B) and (C)

Simplification for Discussion

[0263] The neighboring registers *201* of all the enabled memory elements are collectively referred to as the neighboring layer. The operation registers *200* of all the enabled memory elements are collectively referred to as the operation layer. The data registers *202* of all the enabled memory elements are collectively referred to as the data layers*202*. The status bits and the carry bits of the status registers *203* of all the enabled memory elements are collectively referred to as the status layer and the carry layer, respectively.

[0264] In a database memory or a math 1D memory, the neighboring layers of the memory element whose address is immediately lower or immediately higher than that of the memory element which is being operated on is called the

left layer *114a* and the right layer *114b*, respectively. In a math 2D memory, the neighboring layers of the memory element whose Y address is the same as while whose X address is immediately lower or immediately higher than that of the memory element which is being operated on is called the left layer and the right layer, respectively; while the neighboring layers of the memory element whose X address is the same as while whose Y address is immediately lower or immediately higher than that of the memory element which is being operated on is called the bottom layer and the top layer, respectively.

[0265] If a database memory contains non-addressable registers, the content of its non-addressable is accessible through its operation register *200* and any one of its addressable registers *106*. Thus, all registers are treated as addressable registers *106*. And the operation register *200* should be addressable for optimal performance in this case.

[0266] The following simplifications are applied only for discussing the usage of the CP memory. They are by no mean the constraints on the construct or application of the CP memory.

[0267] Each memory element has only one status bit in its status register *203*. Each of its other registers *200*, *201*, and *202*

has enough bit width to hold each datum for the array.

[0268] Each memory element has only one neighboring register *201*.

[0269] An array of total N items is stored in the data layer(s) *202* of a database memory or a math memory, and the status bits of all memory elements are reset initially. The start address and the end address for the general decoder *110* of the memory are defaulted to point to the first and last items of the array respectively, and the carry number for the general decoder of the memory is defaulted to 1.

USE OF DATABASE MEMORY

[0270] A database memory provides instant execution of almost all basic operations to manage database tables, each of which is an array of records. The following table compares the order of required instruction cycle count of all basic operations using a conventional random access memory (RAM) vs. using a database memory (DBM):

Speed improvement of using database memory

OPERATION	RAM	DBM
Delete any item	~N	~1
Insert a new item	~N	~1
Match an item	~N or log(N)	~1
Count matched items	~N	~1

Enumerate M matched items	~N	~M
Histogram of M sections	~N	~M
Find local max/min	~N	~1
Find global max/min	~N	~log(N)
Order all items	~(N Log(N)) to ~N^2	~sqrt(N) to ~N

[0271] In the above table, for match using RAM, a normal match requires ~N instruction cycles; if a index table has been maintained for the item to be matched, the match is done using binary tree search and it requires ~log(N) instruction cycles.

[0272] In the above table, both the average and the worse-case instruction cycles for ordering all items are given.

USE OF MATH MEMORY

[0273] A math 1D memory can be used instead of a database memory to hold arrays and database tables, providing additional benefit of: (A) counting degree of matching; (B) Find local minimum and maximum using a difference threshold; and (C) provide more efficient sorting algorithm.

[0274] The parallel problems can be solved much more efficiently using a math memory (M1M or M2M) than using a conventional random access memory (RAM). The required instruction cycle counts for most common parallel problems

are shown in the following:

Speed improvement of using 1D math memory

OPERATION	RAM	M1M
Filter of size M	~(N M)	~M
Sum	~N	~sqrt(N)
Match template of size M	~(N M)	~M^2

Speed improvement of using 2D math memory

OPERATION	RAM	M2M
Filter of size (Mx by My)	~(Nx Ny Mx My)	~(Mx My)
Sum	~(Nx Ny)	~cbrt(Nx Ny)
Match template of size (Mx by My)	~(Nx Ny Mx My)	~(Mx^2 My)
Recognize Line (to 1/D angle)	~(Nx Ny D^2)	~D^2

CONTENT MOVING

[0275] An algorithm for deleting the item at a deletion element address is:

[0276] (1) Set the start address to one above the deletion element address.

[0277] (2) Copy a data layer*202* to the operation layer *200*.

[0278] (3) Copy the operation layer *200* to the neighboring layer *201*.

[0279] (4) Set the start address to the deletion element address.

[0280] (5) Set the end address to one below the last used memory element *108* of all the database memory.

[0281] (6) Copy the operation layer *200* from the right layer *114b*.

[0282] (7) Copy the operation layer to the same data layer *202*.

[0283] (8) Repeat step (1) to (7) for all other data layers *202*.

[0284] An algorithm for inserting a new item to an insertion element address is:

[0285] (1) Set the start address to the insertion element address.

[0286] (2) Copy a data layer *202* to the operation layer *200*.

[0287] (3) Copy the operation layer *200* to the neighboring layer *201*.

[0288] (4) Set the start address to one above the insertion element address.

[0289] (5) Set the end address to one above the last used memory element *108* of all the database memory.

[0290] (6) Copy the operation layer *200* from the left layer *114a*.

[0291] (7) Copy the operation layer to the same data layer *202*.

[0292] (8) Repeat step (1) to (7) for all other data layers *200*, to move all the items above the insertion address up by one element.

[0293] (9) Copy a datum of the new item from the data bus of the

external data bus *102* to the corresponding data register *202* of the memory element *108* at the insertion element address using the exclusive bus *105*.

[0294] (10) Repeat step (9) until all the data of the new item are copied from the external data bus *102* to the corresponding data registers *202* of the memory element *108* at the insertion element address.

[0295] Because of its instant content moving ability, a database memory has all the benefit of a content movable memory. The tables stored in the database memory are truly dynamic, without needs for look-ahead allocation and link list, and at the same time the database memory is closely packed, without being fragmented after extensive insertions and deletions. Instead of the element address of the memory element that stores the record, each record can be referred by its primary key ID, and the actual storage of the data may be managed internally by the database memory.

[0296] A math memory has all the benefit of a database memory. Using similar algorithm, a 2D math memory can insert or delete its data based on columns and rows.

CONTENT MATCHING

[0297] An algorithm for matching items is:

[0298] (1) Copy the data layer *202* to be matched to the operation layer*200*.

[0299] (2) Assert positively the status layer.

[0300] (3) Match the operation layer*200* with the data portion *204* of the concurrent bus *109*, according to the "condition" of the concurrent bus *109*, which is the logical opposite of the match requirement, and negatively assert the status layer if the "condition" is met.

[0301] (4) If there are further matching conditions, repeat step (3).

[0302] (5) The matched items have positively asserted status bits, and further operation may be carried out concurrently on the matched items without knowing their actual positions.

[0303] It is easy to design an alternative algorithm similar to the above algorithm based on the match requirement rather than its logical opposite, or combination of the two.

[0304] An algorithm for counting matched items is:

[0305] (1) Assert positively the match bit outputs *112* of all the matched memory elements *108* concurrently.

[0306] (2) The count output of the parallel counter *113* contains the count of the matched memory elements.

[0307] An algorithm for enumerating matched items is:

[0308] (1) Set the priority of the priority encoder *113* to be from high to low.

[0309] (2) Assert positively the match bit outputs *112* of all the matched memory elements*108* concurrently.

[0310] (3) If the no-hit bit output of the priority encoder *113* is positively asserted, all the matched memory elements *108* have been enumerated, and the enumerating algorithm is terminated. Otherwise, the the address output of the priority encoder *113*contains the highest element address of the matched memory elements *108* between the start address and the end address.

[0311] (4) Set the end address to one less than the element address which has been found in step (3).

[0312] (5) Repeat step (3) to step (4).

[0313] It is easy to design an alternative algorithm similar to the above algorithm based on the low-to-high priority of the priority encoder *113*. Due to the design of the general decoder *110*, changing its start address input may be less efficient than changing its end address input.

[0314] Because any matching operation in an array of N items stored in a conventional random access memory requires ~N instruction cycles, traditional databases relies on index tables, each of the index tables stores the sorting order of

a field in the original table. During a match on the field, the index tables are matched using a binary-tree search, requiring ~log(N) instruction cycles, instead of the ~N instruction cycles required when the original table is matched. When a new record is added, or an existing record is modified, the index tables are modified accordingly. The extensively use of index tables requires a lot of additional memory and processing powers. Especially, all index tables have to be updated properly and promptly, otherwise if any index table contains wrong information, the search results become unreliable, and the database itself may become unstable. Managing index tables is a major tasking in any traditional databases.

[0315] When using database memories to store the array, matching items or counting matched items only takes ~1 instruction cycles. This means not only that the required instruction cycles are greatly reduced, but also that the index tables are no longer required, so that the database can be much more efficient and stable.

[0316] The processing power of math memory adds new functionality to the database management. An algorithm for matching items and calculating degrees of matching using a math 1D memory is:

[0317] (1) Send a zero to all the memory elements using the concurrent bus *109* and copy it to the operation layer *200*.

[0318] (2) Match all the memory elements *108* against one requirement.

[0319] (3) Send a weight number to all the memory elements using the concurrent bus and add it to the operation layer *200* of all the matched memory elements *108* in step (2).

[0320] (4) If there are further matching requirements, repeat step (2) to (3) for all the memory elements *108*.

[0321] (5) The operation layer *200* contains the degree of matching of the requirements.

[0322] Similar algorithm can be constructed using a data base memory which can increment its operation layer*200*.

[0323] The ability to calculate the degree of matching not only allows exactly match as currently provided by conventional database engine, but also allows quantified fussy match as currently provided by web search engine. The items of the array may be further handled according to their degree of matching.

CONTENT STATISTICS

[0324] An algorithm to construct a histogram of M sections is:

[0325] (1) Copy the data layer *202* to be matched to the operation

layer *200*.

[0326] (2) Assert positively the match bit outputs *112* of all the memory elements whose status layer is negatively asserted and whose operation layer *200* is larger than the data portion *204* of the concurrent bus *109*, which contains the first section limit from large to small.

[0327] (3) The count output of the parallel counter *113* contains the histogram count of the first section.

[0328] (4) Assert positively the status layer of all the memory elements whose operation layer *200* is larger than the data portion *204* of the concurrent bus *109*. Step (4) masks off those memory elements *108* which have already been counted.

[0329] (5) Repeat step (2) to (4) for all the rest section limits from large to small of the M histogram sections.

[0330] The histogram of the data can be used to estimate the sum and the distribution of the data.

[0331] An algorithm to find the local maximums is:

[0332] (1) Copy the data layer *202* to be characterized to the operation layer *200*.

[0333] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0334] (3) Assert positively the status layer of all the memory ele-

ments each of whose operation layer *200* is larger than the neighboring layer *114* of their neighboring memory elements. This procedure can be carried out in two steps: (A) positively assert the status layer if the operation layer *200* is larger than the neighboring layer *114a* of one of their neighboring memory elements; and (B) negatively assert the status layer if the status layer itself is positively asserted and the operation layer *200* is smaller than the neighboring layer *114b* of any other of their neighboring memory elements.

[0335] An algorithm to find the local minimums can be similarly constructed.

[0336] An algorithm to find the local maximums with a difference threshold using a math memory is:

[0337] (1) Copy the data layer *202* to be characterized to the operation layer *200*.

[0338] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0339] (3) Send the difference threshold to all the memory elements *108* through concurrent bus *109* and add it to the operation layer*200*.

[0340] (4) Assert positively the status layer of all the memory elements each of whose operation layer *200* is larger than the

neighboring layer *114* of their neighboring memory elements.

[0341] The algorithm to find the local minimums using difference threshold is similar.

[0342] The use of difference threshold reduces the effect of noise presented in the original data when determining the local minimums and maximums.

[0343] An open-end binary tree algorithm to find a global upper limit to the data is:

[0344] (1) Designate a variable denoted as FOLD, and initiate it with 1.

[0345] (2) Designate a variable denoted as ADDR.

[0346] (3) Designate a variable denoted as MAX.

[0347] (4) Designate a variable denoted as VAL.

[0348] (5) Find the local maximums. As a result, the memory elements *108* which are local maximums have status layer positively asserted and operation layer *200* containing data to be characterized.

[0349] (6) Set the priority of the priority encoder *113* to be from high to low.

[0350] (7) Assert positively the match bit outputs *112* of all the memory elements *108* whose status layer has been posi-

tively asserted.

[0351] (8) Read the output address of the priority encoder *113* into ADDR.

[0352] (9) Read the operation register *200* in the memory element *108* at element address ADDR into MAX.

[0353] (10) Let VAL = MAX.

[0354] (11) Set the end address to be one less than ADDR.

[0355] (12) Assert positively the match bit outputs *112* of all the memory elements *108* whose status layer has been positively asserted and whose operation layer *200* is larger than VAL.

[0356] (13) Read the output address of the priority encoder *113*. If it contains NULL, a global upper limit is in VAL while the largest known value is MAX at the element address ADDR, and the algorithm terminates. Otherwise, save it into ADDR.

[0357] (14) Read the operation register *200* in the memory element *108* at the element address ADDR into MAX.

[0358] (15) Let VAL = VAL + (MAX - VAL) * FOLD.

[0359] (16) Double the value of FOLD.

[0360] (17) Repeat step (11) to (16).

[0361] The algorithm can be continued in close-end binary tree manner to find the global maximum of the data, as the following:

[0362] (20) Let VAL = (VAL + MAX) / 2.

[0363] (21) Set the end address to be one less than ADDR.

[0364] (22) Assert positively the match bit outputs *112* of all the memory elements *108* whose status layer has been positively asserted and whose operation layer *200* is larger than VAL.

[0365] (23) Read the output address of the priority encoder *113*. If it contains NULL, VAL contains a better global upper limit, and step (20) to (23) is repeated. Otherwise, save it into ADDR.

[0366] (24) Read the operation register *200* in the memory element *108* at the element address ADDR into MAX.

[0367] (24) Assert positively the match bit outputs *112* of all the memory elements whose status layer has been positively asserted and whose operation layer *200* is larger than MAX.

[0368] (25) Read the output address of the priority encoder *113*. If it contains NULL, the global maximum is MAX at address ADDR, and the algorithm terminates. Otherwise, save it

into ADDR and repeat step (24) to (26).

[0369] To find a upper global limit and the global maximum for a randomly arranged set of {1, 2, 3, ... N}, the above algorithms take ~log(N) instruction cycles on average.

[0370] An algorithm to find a lower global limit and the global minimum can be similarly constructed.

[0371] The ability to quickly find and count the local extreme values, and the ability to quickly find the global limits, the global extreme values, the histogram, the estimated sum of a large set of data means that a database memory or a math memory can be used in statistical processing such as estimating its distribution.

SORTING

[0372] Because of its instant match finding ability, a database memory require no index table and much less sorting of data. Still, it is possible to sort data using a database memory with much less instruction cycles than what is required using a conventional random access memory.

[0373] An algorithm to see if the array is in order is the following:

[0374] (1) Copy the data layer *202* which contains the data to be characterized to the operation layer *200*.

[0375] (2) Copy the operation layer *200* to the neighboring layer

201.

[0376] (3) Set the start address to one more than the first item.

[0377] (4) Assert positively the match bit outputs *112* of all the memory elements *108* whose operation layer *200* is smaller than left layer *114a*.

[0378] (5) Read the count output of the parallel adder *113*. If it equals zero or the total item count, the array is already sorted.

[0379] The above count output is the disorder count to sort the array into small-to-large order. The disorder count to sort the array into large-to-small order can be found similarly. To sort an array in either way is functionally equivalent--the other sorting order can be achieved by reading from end item to start item of one sorting order. Thus, the two disorder count is compared to select a better sorting order of the two, and the worst case for sorting--to sort an almost sorted array into another order--can be avoided.

[0380] There are two ways to disorder an already ordered array: (A) to randomly exchange the adjacent neighboring items, to create local disorder; or (B) to remove and insert an item randomly to another location, to create global disorder. These two kinds of disorders are dealt with by local exchange sorting algorithm and global moving sorting al-

gorithm respectively.

[0381] A local exchange sorting algorithm concurrently exchanges all the adjacent two items into correct order. An algorithm to exchange once the adjacent even and odd numbered items toward small-to-large order is:

[0382] (1) Carry out the algorithm to find out the disorder count. If it is zero, the sorting algorithm terminates. As a result, both the operation layer *200* and the neighboring layer *201* contain data to be characterized.

[0383] ((2) Set the carry number to 2.

[0384] (3) Set the start address to one more than the first item.

[0385] (4) Copy the operation layer *200* from the left layer *114a* if the latter is larger.

[0386] (5) Exchange the operation layer *200* and the data layer *202* to be characterized if they are different.

[0387] (6) Set the start address to the first item.

[0388] (7) Set the end address to one less than the last item if the total item count is odd.

[0389] (8) Copy the operation layer *200* from the right layer *114b* if the latter is smaller.

[0390] (9) Exchange the operation layer *200* and the data layer *202* to be characterized if they are different. If each item

contains only one data layer *202*, the algorithm terminates.

[0391] (10) Assert positively the status layer when the operation layer *200* and the data layer *202* to be characterized are different.

[0392] (11) Copy one of the other data layers *202*, which is the data layer to be transferred, to the operation layer *200*.

[0393] (12) Copy the operation layer *200* to the neighboring layer *201*.

[0394] (13) Set the start address to one more than the first item.

[0395] (14) Set the end address to the last item if the total item count is odd.

[0396] (15) Copy the operation layer *200* from the left layer *114a* if the status layer is positively asserted.

[0397] (16) Exchange the operation layer *200* and the data layer *202* to be transferred.

[0398] (17) Copy the operation layer *200* to the neighboring layer *201*.

[0399] (18) Set the start address to the first item.

[0400] (19) Set the end address to one less than the last item if the total item count is odd.

[0401] (20) Copy the operation layer *200* from the right layer *114b*

if the status layer is positively asserted.

[0402] (21) Copy the operation layer *200* to the data layer *202* to be transferred.

[0403] (22) Step (11) to (21) exchanges the data layer *202* to be transferred of the adjacent even and odd numbered items which need to be exchanged. Repeat step (11) to (21) to transfer each of all the other data layers.

[0404] An example of such sorting of a one-layer array is the following:

(1)

Data Layer	5	4	3	2	2	2	6	1
Operation Layer	5	4	3	2	2	2	6	1
Neighbor Layer	5	4	3	2	2	2	6	1

(4)

Data Layer	5	4	3	2	2	2	6	1
Operation Layer	5	5	3	3	2	2	6	6
Neighbor Layer	5	4	3	2	2	2	6	1

(5)

Data	5	5	3	3	2	2	6	6

Layer								
Opera-tion Layer	5	4	3	2	2	2	6	1
Neighbor Layer	5	4	3	2	2	2	6	1

(8)

Data Layer	5	5	3	3	2	2	6	6
Opera-tion Layer	4	4	2	2	2	2	1	1
Neighbor Layer	5	4	3	2	2	2	6	1

(9)

Data Layer	4	5	2	3	2	2	1	6
Opera-tion Layer	5	4	3	2	2	2	6	1
Neighbor Layer	5	4	3	2	2	2	6	1

[0405] An algorithm to exchange the adjacent odd and even numbered items once into small-to-large order can be similarly constructed. The local exchange sorting algorithm comprises of repeated alternative execution of algorithm to exchange once the adjacent items of (A) even and odd numbered, and (B) odd and even numbered. Us-

ing this sorting algorithm alone can sort an array in no more than ~N instruction cycles.

[0406] The local exchange sorting algorithm may not be efficient enough because the array may be nearly sorted except a very few difficult items which still walk one element at a time toward their final destinations. Using a math 1D memory, or a database memory which can increment or decrement its operation layer, an improved algorithm awards walks in correct direction with jumps:

[0407] (1) A minimal and a maximal cap of the array are inserted as the first and last items, respectively.

[0408] (2) Each item carries a walk number, which is initiated to 0.

[0409] (3) Designate a threshold M for the walk number.

[0410] (4) Check if the array is already sorted. If yes, terminate the algorithm by removing the first and last items.

[0411] (5) Carry out one local exchange sorting toward small-to-large order. If an item walks to right, its walk number is increased by 1; otherwise, if it walks to left, its walk number is decreased by 1.

[0412] (6) By content matching means, with the priority from right to left, enumerate an item whose walk number reach +M.

[0413] (7) The value of each such item is compared with all the other items to its right using content matching means, with the priority from left to right, and the leftmost item whose value is not smaller than such item is found.

[0414] (8) If the newly found item has not a negative walking number, such item is moved to the left of the newly found item. The walk number of such item is reset to 0.

[0415] (9) By content matching means, with the priority from left to right, enumerate an item whose walk number reach -M.

[0416] (10) The value of each such item is compared with all the other items to its left using content matching means, with the priority from right to left, and the rightmost item whose value is not larger than such item is found.

[0417] (11) If the newly found item has not a positive walking number, such item is moved to the right of the newly found item. The walk number of such item is reset to 0.

[0418] (12) Repeat step (6) to (11) until there is no item whose walk number reaches either +M or -M.

[0419] (13) Repeat step (4) to (12).

[0420] To order a randomly arranged set of {1, 2, 3 ... N}, when M equals to sqrt(N), the above algorithm takes ~sqrt(N) instruction cycles on average.

[0421] A global moving sorting algorithm removes disordered

items in a nearly sorted array and inserts them to proper place. It does this by analyzing "topography" of the sorting disorders. Peak and valley are used to describe sorting disorder. A peak *331* is an item whose data layer to be characterized contains value larger than those of its both neighbors', while a valley *341* is an item whose data layer to be characterized contains value smaller than those of its both neighbors'. For the small-to-large sorting order, a true valley or a true peak has right neighbor not smaller than left neighbor. Otherwise they are false valley or false peak respectively. General cases of sorting disorder are shown in FIG. 19, which shows:

[0422] (1) Single true valley: Case *321* is identified by a true valley *342*with an adjacent false peak *332* to the left. When the true valley *342* is removed, the false peak *332* disappears also.

[0423] (2) Single true peak: Case *322* is identified by a true peak *333* with an adjacent false valley *343* to the right. When the true peak *333* is removed, the false valley *343* disappears also.

[0424] (3) A section of data which is ordered in incorrect order: Case *323* is identified by a lone true peak *334* to its left, and a lone false valley *344* to its right. Case *324* is identi-

fied by a lone false peak *335* to its left, and a lone true valley *345* to its right. Case *323* and *324* can be merged together, with a lone true peak *334* to its left, and a lone true valley *345* to its right. Remove one true peak or valley from the end of any sections generates another true peak or valley, until the whole section is removed. Any of these sections may contain lone pairs of apparently false valley with an adjacent apparently true peak to the right, or lone pairs of apparently true valley with an adjacent apparently false peak to the right. Because the topography is reversed from that of single true valley or peak, the apparently false valley or peak is actually true, while the apparently true valley or peak is actually false.

[0425] (4) A section of data which is ordered in correct order but in incorrect increment: Both Case *325* and *326* are identified by an adjacent pair of false peak and false valley, as *336* and *346*, and *337* and *347*. Case *325* and *326* can merge together. Applying a local exchange sorting algorithm separates out either a true peak or a true valley or both, from the ends of the sections. Any of these sections may contain a single true valley or peak within the section.

[0426] The leftmost true valley item can be moved to the right of the first item to its left which is smaller than it, or to the

left end of the array, in ~1 instruction cycles. The right-most true peak item can be moved to the left of the first item to its right which is larger than it, or to the right end of the array, in ~1 instruction cycles. Applying these two procedures is the global moving sorting algorithm, which may also be used between the applications of local exchange sorting algorithm to accelerate the sorting.

Local Operations

[0427] The connectivity and arithmetic ability of a math memory enables local operations, such as filtering. A local operation involving M neighbors takes ~M instruction cycles generally, independent of the total array item count N.

[0428] For simplicity of following discussion, the neighboring layer *201* contains the data to be characterized, or the content of the memory element. A special 1D vector of odd-number of items is used to describe the content composition of the operation layer *200* of all the enabled memory elements after a concurrent 1D local operation in a 1D math memory. The center item describes the content originated from the element itself and is indexed as 0. The item left to the center item describes the content originated from the left neighbor of the element and is indexed as -1. The item right to the center item describes

the content originated from the left neighbor of the element and is indexed as +1. So forth. For an example, (1) denotes the content of all the enabled memory elements; (1 0 0) denotes the content of left neighbors to all the enabled memory elements; (1 1 0) denotes adding the content of left neighbors to the content of all the enabled memory elements; and (1 1 1) denotes three point average for all the enabled memory elements.

[0429] Two successive operations can be additive if both of them use the operation layer accumulatively, such as:

[0430] (1 1 0) = (1) + (1 0 0);

[0431] Mathematically, a + operation is defined as:

[0432] C = A + B: C[i] = A[i] + B[i];

[0433] The + operation satisfies:

[0434] A + B = B + A;

[0435] (A + B) + C = A + (B + C);

[0436] When the operation layer *200* is copied to or exchanged with the neighboring layer *201*, the successive operations are no longer addictive. For example, a 3-point (1 2 1) Gaussian averaging algorithm is:

[0437] (1) Copy the data layer *202* to be averaged to the opera-

tion layer *200*.

[0438] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0439] (3) Set the start address to be one more than the first item.

[0440] (4) Set the end address to be one less than the last item.

[0441] (5) Add the left layer *114a* to the operation layer *200*.

[0442] (6) Copy the operation layer *200* to the neighboring layer *201*.

[0443] (7) Add the right layer *114b* to the operation layer*200*. The result is in the operation layer.

[0444] In the above algorithm, the additive result of step (2) and (5) is subjected to Step (7) due to Step (6). Without step (6), step (7) is also additive to step (2) and (5), and the algorithm result is (1 1 1). When the result of a first operation A undergoes a second operation B, the overall operation C is expressed mathematically as:

[0445] $C = A \# B$: $C[i] = \Sigma_j (A[i+j]\ B[i-j])$;

[0446] The # operation satisfies:

[0447] $A \# B = B \# A$;

[0448] $(A \# B) \# C = A \# (B \# C)$;

[0449] The # and + operations satisfy:

[0450] (A + B) # C = (A # B) + (A # C);

[0451] The 3-point (1 2 1) Gaussian averaging algorithm is expressed as:

[0452] (1 2 1) = (1 1 0) # (0 1 1);

[0453] A 5-point Gaussian averaging is:

[0454] (1 2 4 2 1) = (1 1 1) # (1 1 1) + (1);

[0455] The corresponding algorithm can be read from the mathematical expression, as:

[0456] (1) Copy the data layer *202* to be averaged to the operation layer *200*.

[0457] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0458] (3) Set the start address to be one more than the first item.

[0459] (4) Set the end address to be one less than the last item.

[0460] (5) Add the left layer *114a* to the operation layer *200*.

[0461] (6) Add the right layer *114b* to the operation layer *200*. Step (4) to (6) carry out the first (1 1 1) operation.

[0462] (7) Exchange the operation *200* and the neighboring layers *201*. Step (7) carries out the # operation.

[0463] (8) Add the left layer *114a* to the operation layer *200*.

[0464] (9) Add the right layer *114b* to the operation layer *200*. Step (7) to (9) carry out the second (1 1 1) operation.

[0465] (9) Add the neighboring layer*201* to the operation layer *200*. Step (9) carries out the "+ (1)" operation.

[0466] This concept is extendable to 2D local operations, such as a 9-point Gaussian averaging:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = (1 \quad 1 \quad 0) \,\#\, (0 \quad 1 \quad 1) \,\#\, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \,\#\, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix};$$

[0467] The corresponding algorithm can be read from the mathematical expression, as:

[0468] (1) Copy the data layer *202* to be averaged to the operation layer *200*.

[0469] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0470] (3) Set the start X address to be one more than the left boundary.

[0471] (4) Set the end X address to be one less than the right boundary.

[0472] (5) Set the start Y address to be one more than the bottom boundary.

[0473] (6) Set the end Y address to be one less than the top boundary.

[0474] (7) Add the left layer to the operation layer *200*.

[0475] (8) Copy the operation layer *200* to the neighboring layer *201*.

[0476] (9) Add the right layer to the operation layer *200*.

[0477] (10) Copy the operation layer *200* to the neighboring layer *201*.

[0478] (11) Add the bottom layer to the operation layer *200*.

[0479] (12) Copy the operation layer *200* to the neighboring layer *201*.

[0480] (13) Add the top layer to the operation layer*200*.

[0481] Or a 9-point 0-degree Sober filtering:

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = (-1 \quad 0 \quad 1) \# \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \# \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix};$$

[0482] The corresponding algorithm can be read from the mathematical expression, as:

[0483] (1) Copy the data layer *202* to be characterized to the operation layer *200*.

[0484] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0485] (3) Set the start X address to be one more than the left boundary.

[0486] (4) Set the end X address to be one less than the right boundary.

[0487] (5) Set the start Y address to be one more than the bottom boundary.

[0488] (6) Set the end Y address to be one less than the top boundary.

[0489] (7) Copy the left layer to the operation layer *200*.

[0490] (8) Negate the operation layer *200*.

[0491] (9) Add the right layer to the operation layer *200*.

[0492] (10) Copy the operation layer *200* to the neighboring layer *201*.

[0493] (11) Add the bottom layer to the operation layer *200*.

[0494] (12) Copy the operation layer *200* to the neighboring layer *201*.

[0495] (13) Add the top layer to the operation layer *200*.

SUM

[0496] To sum a one-dimensional array of N items, the array is divided into sections, each of which contains M consecutive items. All sections are summed concurrently from left to right, in ~ M instruction cycles. Then the section sums, which are at the right-most items of every sections, are

summed together serially in ~ N / M instruction cycles. Thus, the total instruction cycle count is ~(M + N / M). When M ~ sqrt(N), the total instruction cycle count has a minimum of ~ sqrt(N). A detailed sum algorithm is:

[0497] (1) Copy the data layer *202* to be summed to the operation layer *200*.

[0498] (2) Copy the operation layer *200* to the neighboring layer *201*.

[0499] (3) Set the carry number to M ~ sqrt(N). The M is the item count in each section, except the last section which may have items less than M.

[0500] (4) Increment the start address by one.

[0501] (5) Add the left layer *114a* to the operation layer *200*.

[0502] (6) Exchange the operation layer *200* and the neighboring layer *201*.

[0503] (7) Repeat step (4) to (6) M times. The section sums are at the neighboring layer *201* of the last items of all sections.

[0504] (8) Read and add all the neighboring registers *201* of all last items of all sections serially, to get the sum of the array.

[0505] For an example, an array starts with (0, 1, 2, 3, 4, 5, 6, 7) is summed as:

Example of 1D array summing

step	operation layer	neighboring layer
2	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 2, 3, 4, 5, 6, 7
3	M = 3	
5a	0, 1, 2, 3, 7, 5, 6, 13	0, 1, 2, 3, 4, 5, 6, 7
6a	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 2, 3, 7, 5, 6, 13
5b	0, 1, 3, 3, 4, 12, 6, 7	0, 1, 2, 3, 7, 5, 6, 13
6b	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 3, 3, 4, 12, 6, 13
	accumulator	
8a	3	
8b	+ 12 = 15	
8c	+ 13 = 28	

[0506] The above algorithm can be displayed by an algorithm flow diagram in FIG. 20, in which serial operations are represented by a series of simple arrows *351*, and concurrent parallel operations are represented by a series of arrow with two parallel bars on each side *352*. Each arrow shows the data range of the operation, such as on a section *356* with M items of the whole array *357* with N items. Each series of arrows is marked by a step sequence number followed by ":" *353*, an instruction cycle count preceded by "~" *354*, and an operation *355*. The instruction cycle counts from consecutive and independent steps are

additive, so the total instruction cycle count is (M + N / M) >= ~ sqrt(N), which has a minimum of ~sqrt(N) when M ~ sqrt(N).

[0507] To sum a two-dimensional array of Nx by Ny items, the array is divided into sections, each of which contains Mx by My consecutive items. All rows of all sections are summed concurrently from left to right, in ~ Mx instruction cycles. Then all the right-most columns of all sections, each item of which contains a row sums for a section, are summed concurrently from bottom to top. Then the top-right-most items of all sections, each of which contains a section sum, are scanned and summed together serially, with the column and the row direction being the fast and the slow scan direction respectively. FIG. 21 is the corresponding algorithm flow diagram, in which step sequence number "4 * 3" *358* means a complete step 3 is carried out before each instruction cycle of step 4. Thus, the total instruction cycle count for such combination of steps is the product of the individual instruction cycle count of each step. The total instruction cycle count is (Mx + My + Nx / Mx Ny / My), which has minimum of cbrt(Nx Ny) when Mx ~ My ~ cbrt(Nx Ny).

TEMPLATE MATCHING

[0508] To match a template of size M, the array is divided into N / M sections, each of which contains M consecutive items. The algorithm diagram is shown in FIG. 22. In Step 1, the template to be matched is loaded to all sections concurrently in ~M instruction cycles. Then the point-to-point absolute difference is calculated concurrently for all points in ~ 1 instruction cycles, which is omitted from the algorithm flow diagram. In Step 2, all sections are summed concurrently from right to left in ~ M instruction cycles, to obtain the difference values of the array to the pattern at the first positions to the left of all sections. In the first instruction cycle of Step 3 * 2, the templates in all sections are shifted right by one item, to calculate the difference at the second positions of all sections, and so forth. Thus the total instruction cycle count is (M + M M) ~ M^2. When M > sqrt(N), the summing of all sections is further divided into the concurrent summing of subsections, each of which contains L consecutive items, and the serial summing of the subsections, thus the total instruction cycle count is ~(M + (L + N / L) M), or ~ (M sqrt(N)) when L ~ sqrt(N).

[0509] Similar algorithm can be carried out in a 2-D array of size Nx by Ny stored in a math 2D memory for a 2-D template

of size Mx by My. The algorithm diagram is shown in FIG. 23, which also omits the step of calculating point-to-point absolute difference. In step 2 *1, the template to be matched is loaded to all sections concurrently in ~M^2 instruction cycles. The first application of Step 3 sums the point-to-point absolute differences of each of all section at the first column from left of the section. The first instruction cycle of Step 4 moves the template right by one column. The second application of Step 3 sums the point-to-point absolute differences of each of all sections at the second column from left of the section. The first complete application of Step 4 * 3 fills the sums of row difference of the corresponding section. The first application of Step 5 results in the matching of the template at the first row from bottom of each of all sections. The first instruction cycle of Step 6 * (4 * 3 + 5) moves the template up by one row. The Step 4 * 3 is carried out again except that the Step 4 is carried out from right to left this time, since the first application of the Step 4 *3 has moved the template to the right-most position of each section. Thus, the total instruction cycle count is ~(Mx My + (Mx^2 + My) My), which is equivalent to ~(Mx^2 My).

[0510] Using CP memory, the instruction cycle count for 1-D

template matching is reduced from ~ (N M) to ~M^2, and it is reduced from ~(Nx Ny Mx My) to ~(Mx^2 My). It may be small enough now for the template matching algorithm to be carried out in real-time for a lot of applications, such as image database.

THRESHOLDING

[0511] With its multiple dimensions of data, image processing and spatial modeling generally requires large amount of calculation, which is linearly proportional to the size of data in each dimension.

[0512] Using a conventional bus-sharing computer, the instruction cycle count is linearly proportional to the amount of calculation. Thus, to solve a problem in a realistic time period, thresholding is frequently used to ignore large amount data in the subsequent processing. Thresholding is a major problem, because proper thresholding is difficult to achieve, and thresholding in different stages may interact with each other.

[0513] Using a math memory, the instruction cycle count is decoupled from the amount of calculation, and is independent of the size of data in each dimension. Thus, thresholding can be used only in last stage to qualify the result. Also, thresholding itself has been reduced to ~1 instruc-

tion cycle operation.

[0514] For an example, to recognize features of an image, one of the common conventional methods is to:

[0515] (1) Use Sobel filters to find edge intensity of the image.

[0516] (2) Use thresholding to ignore most pixels except those which have large edge intensities. In most practice the image is further reduced into a binary bitmap.

[0517] (3) Analyze the reduced data set for features, such as carry out line recognition.

[0518] In step (2), if the threshold is too high, true edge pixels may be ignored. On the other hand, if it is too low, none edge pixels may be included. Both cases add difficulty to step (3) and subsequent analysis. If the illustration of the image is not uniform, or the image contains features of different reflectivity, or the objects cast shadows, it is almost certain that there is no perfect global threshold for edge intensity, and thresholding process itself may become very complicated.

[0519] Using a math memory, step (1) and (2) can be altogether canceled, and the raw intensities of all pixels are used for subsequent processing without any increase of instruction cycles. Thresholding may be applied to visualize the processed image after a step, but it can be kept out of the

image processing itself until the last step.

Line Recognition

[0520] Due to neighbor-to-neighbor connectivity, CP memory can treat line detection problem as a neighbor counting problem. A line can be made of pixels of up to a distance apart, which is called the pixel span of the line. A continuous line lying exactly along X or Y direction thus has pixel span of 1. On a real image, edge lines are of primary importance, each of which separates pixels on its two sides into two intensity groups. Thus, the following discussion is limited to detecting edge lines, although the stated algorithms can be easily adopted to detecting other lines, such as intensity lines.

[0521] To detect edges line of pixel span 1 and pixel length L lying exactly along X direction left to each pixel, the neighborhood count algorithm is direct:

[0522] (1) Each of all pixels subtracts the raw intensity of its bottom layer from that of its top layer, and stores the result in the neighboring layer.

[0523] (2) Each of all pixels sums the neighboring layers of its L left neighbors together with its own. The absolute value of the result indicates the possibility of an edge line starting from that pixel, while the sign of the result indicates

whether the edge is rising or falling along the Y direction.

[0524] The algorithm to detect edge line lying exactly along X direction is similar.

[0525] To detect edge lines with a slope of (My / Mx), each pixel defines a super lattice of Mx by My pixels denoted as (Mx * My), and the line which connects the pixel and the furthest corner of the super lattice has the slope of (My / Mx). Similar to obtaining the section sums in a sum algorithm, a messenger starts from furthest corner of the super lattice, walks (Mx + My) steps along the line until it reaches the original pixel. In each of its stop, if the pixel is on the left side of the line, its intensity is added to the messenger; otherwise, if the pixel is on the right side of the line, its intensity is subtracted from the messenger. When reaching the original pixel, the value of the messenger indicates the possibility and the slope of the edge line which connects the original pixel and the furthest corner of the (Mx * My) super lattice. Similar process may carry out for the (-Mx * -My) super lattice. This accumulating process is carried out concurrently for all the pixels of the image, independent of image sizes. FIG. 24 shows the (4 * 3) super lattice to detection a line with a slope of (3/4) passing the original pixel at *0*. The accumulation process-

ing is from pixel *7* to pixel *0* in sequence, with the intensities of pixel *1*, *3*, and *5* to be added, and the intensities of pixel *2*, *4*, and *6* to be subtracted from the messenger.

[0526] If multiplication is used, the line detection algorithm can be further improved. At each stop of the line detection algorithm, through the concurrent bus *109*, a weight factor for the stop is sent concurrently to all the messengers, which multiply the weight factor with the pixel intensities of the stop and accumulate the result. The weight factor is inversely proportional to the distance between the line and the pixel at the stop. In FIG. 24, assuming the edge line half-width of 1, the corresponding weight factors could be:

An example of weight factors for line detection

Pixel	1	2	3	4	5	6
Width	+2/5	-4/5	+3/5	-3/5	+4/5	-2/5

[0527] Given an angular resolution requirement, a {(Mx, My)} set can be constructed to detect all lines on an image, each element of which can be determined by a corresponding line detection algorithm. FIG. 25a shows a set of origin-bounding lines whose pixel spans are exactly 7 in walking distance, on a square grid. It also shows the walking distance envelope of 7. For such a line set of walking dis-

tance D, the angular resolution is ~(2/D) along the 45-degree diagonal direction, and ~(1/D) along the X and Y directions; the total instruction cycle count is ~ D^2, independent of the image size.

[0528] To reduce the instruction cycles for detecting the lines, starting from a {(Mx, My)} set of D in walking distance, a circuit of radius ~(D/sqrt(2)) in real distance may be used to guide the starting walking pixels for the messengers, to also have slightly more uniform angular resolution. FIG. 25b shows such a set of origin-bounding lines whose pixel spans are ~5 in real distance, on the same square grid. It also shows the real distance envelope of 5.

[0529] If a (Mx * My) super lattice in the set have stop(s) that passes the line exactly, lines of short pixel span in that direction also need to be added to the set. For an example, the super lattice (5 * 0) of the set adds super lattices (4 * 0), (3 * 0), (2 * 0) and (1 * 0) to the set. As a result of line detection, each pixel is marked by the line value of the highest normalized absolute value together with its corresponding super lattice.

Long Range Connectivity

[0530] Adding long-range connectivity generally reduces the instruction cycle count for global operations. FIG. 26 shows

the log3(N) long range connectivity, in which N equals 27. In FIG. 26, all dots in each column represent a memory element, which is marked by the element address at the top, and different layer represents different range of connectivity, such as neighbor-to-neighbor or between every 3^0 neighbors *171*, between every 3^1 neighbors *172*, between every 3^2 neighbors *173*, and between every 3^3 neighbors *174*. In limit finding or sum, the results of three neighbors are sent to next layer of longer range of connectivity, so that the total instruction cycle count are log3(N) in both cases. Similar algorithm may also be applied to sorting and fast Fourier transformation.

SUPER-LATTICE CONNECTIVITY

[0531] Long-range connectivity is a special type of super-lattice connectivity. It may be difficult to change the connectivity after a CP memory has been made, but it is quite feasible that all elements in an M-dimension lattice is a subset of a (M+1)-dimension lattice, with each M-dimension lattice connected on a different super lattice.

[0532] FFIG. 27a shows an example of 2-D super-lattice connectivity. Instead of connecting all nodes along the X and Y directions, to detection a line which lies specifically along the direction from node *0* to node *7*, it connects node *0*

and node *7*, and node *0* and node *2*, so that the direct neighborhood counting algorithm can be used concurrently on all the nodes to detect the line in the specific direction. FIG. 27b shows an example of 2-D super-lattice connectivity. It composes of planes of 2-D super-lattice connectivity, each is specialized for detecting lines in one direction similar to that of FIG. 27a. All these planes have same pixel registry between the planes, to allow direct connections between registered nodes between different planes. The image data may come from a steady source, such as a video camera. The data pass all 2-D super lattices in turn, which works concurrently and continuously on the same instructions as part of a SIMD pipeline, and finally emerges with the best line value and the associated super lattice attached to each pixel.

PARALLEL DIVIDER

[0533] FIG.28 shows a circuitry algorithm for parallel divider using an all-line decoder, a carry pattern generator, a parallel counter, and a priority encoder. The dividend *161* is input into an all-line decoder, to generate continuous bit outputs up to the dividend *163*. The divisor *162* is input into a carry-pattern generator, to generate the corresponding carry pattern *164*. The two sets of bit outputs

are AND-combined together. The combined bit outputs are counted by a parallel counter, to get the quotient of the division *165*. Meanwhile, the combined bit outputs are also processed by an encoder of high-to-low priority, to get the largest bit output of the carry pattern generator which is less than or equal to the dividend*166*, and thus the value of dividend minus reminder *167*

[0534] Because a CP memory may already have an all-line decoder, a carry pattern generator, and a parallel counter, by caching the bit outputs of the general decoder, the CP memory may also be a parallel divider in addition, which, due to the functionality of the general decoder, provides slightly more powerful functionality of obtaining the quotient and the value of dividend minus reminder, of dividing a dividend by a divider, the dividend being the value of a subtrahend minus an offset.

Functional Overview of Concurrent Processing memory

[0535] As illustrated by FIG. 29, in which the general decoder, the priority encoder, and the parallel counter have been packed into an element control unit, a general CP memory can be summarized by the following rules:

[0536] (1) A CP memory is made of identical elements, each of which has a unique address.

[0537] (2) Each memory is connected with a data bus.

[0538] (3) One element can read from or write to the data bus exclusively.

[0539] (4) Multiple elements can be activated by an element control unit. A memory element is activated if its element address corresponds to the increments of a carry number starting at a start address and if it is equal to or less than an end address

[0540] (5) Multiple activated elements can read from the data bus concurrently.

[0541] (6) Multiple activated elements can be required to identify themselves concurrently. Each element positively asserts a line which connects the element back to the element control unit

[0542] (7) Each element contains a fixed number of registers.

[0543] (8) The neighboring elements are connected so that an element can read at least one register of its neighbor.

[0544] (9) There is an extra external command pin to indicate the address and data bus contains whether an instruction, or address and data for the memory when it is enabled.

[0545] Rule (1), (2) and (3) specifies the functional backward compatibility with a conventional random access memory.

Rule (4), (5) and (6) defines concurrency. Rule (7) and (8) defines connectivity. Rule (9) defines processing capability.